INCREMENTAL PROCESSING AND OPTIMIZATION OF UPDATE STREAMS

Mengmeng Liu

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2016

Supervisor of Dissertation

Signature_____

Zachary G. Ives

Professor of Computer and Information Science

Co-supervisor of Dissertation

Signature_____

Boon Thau Loo

Associate Professor of Computer and Information Science

Graduate Group Chairperson

Signature_____

Lyle Ungar, Professor of Computer and Information Science

Dissertation Committee

Susan B. Davidson, Professor of Computer and Information Science (Chair)

Val Tannen, Professor of Computer and Information Science

Andreas Haeberlen, Assistant Professor of Computer and Information Science

Tyson Condie, Assistant Professor of Computer Science, University of California, Los Angeles (External)

INCREMENTAL PROCESSING AND OPTIMIZATION OF UPDATE STREAMS

To Zhuowei, Jeremy, and my parents Xindong and Lihua

Acknowledgements

This dissertation is not possible without the help from my advisors, collaborators, friends and family. I am deeply indebted to those individuals who support me throughout the years.

First of all, I would like to thank my advisors, Zachary Ives and Boon Thau Loo, who offered me the opportunity to pursue graduate studies at Penn. Prof. Zachary Ives led me into the field of database system research, and from him I learned tons of research ideas which can even be traced back to some database research several decades ago. He is generally knowledgable in most areas of database systems, and since he has tremendous experience implementing various systems firsthand himself, he is always a great resource to consult for solving problems encountered in practice. His knowledge and vision greatly shapes my taste for database system research, and later on, when I conducted research and development in industrial labs and companies, I can always recall the various conversations I had with him, and use those ideas as a foundation to see how they can be applied to different systems. Zack's pursuit of perfection has a profound influence on me, not only in doing research but also everything in life. Every paper that we wrote together, every experiment that we demonstrated, and every presentation I made throughout the years, underwent numerous revisions with him, and in retrospect, each time I learned something new and pushed myself harder the next time around. Through these endeavors, I learned how to effectively implement an idea, how to present an experimental result, how to make the paper clear, and how to make the presentation intuitive and to the point. All of these experiences tremendously helped me in not only research skills but also almost every aspect in my professional life.

Prof. Boon Thau Loo is not only a great mentor, but also a good friend. I am always

amazed to see how deeply he cares about the student, sometimes it feels like the students are also his own kids. He would go all the way from discussing research to making arrangements and plans for the benefit of the students. His encouragement makes a student feel secure and confident, even through difficult times such as having a paper rejected, or going through difficult things in life. Although he has a busy schedule, he always makes time to meet with the students, listen to their concerns, provide suggestions and help them grow. His knowledge in declarative networks and datalog helped shape and improve my initial research ideas, and I am glad to have the opportunity to work with him on many papers and presentations together throughout the years.

I would also like to thank the other fantastic professors in the Penn DB group, Prof. Susan Davidson and Prof. Val Tannen. We have weekly DB group seminars and throughout the years I learned many great ideas from the seminar, and more importantly, it helped shape my general sense of database research other than my main research area. Prof. Susan Davidson is a great mentor in bridging the gap between theory and practice in database research, and her recommendations in making papers and presentations intuitive and clear have always had a great influence on me. Prof. Val Tannen is an expert in database theory, and from him I learned the relationship between database theory and logic, the foundation of datalog, provenance, as well as incremental view maintenance. He has both a great taste of research and a great sense of humor, and his pursuit of precision in formulating research problems has a great influence on me as well.

Besides the school work, I also did two fruitful summer internships at Microsoft Bing and IBM Research - Almaden respectively. I would like to thank my mentor, Jingren Zhou at Microsoft, who guided me into the field of big data systems in large-scale distributed environment deployed real-life in industry. The internship project on optimizing multiple scans of data streams on top of their big data system, SCOPE and Cosmos, has a great influence not only on my research but also on my career later. I would also like to thank my mentor, Ioana Stanoi at IBM Research - Almaden, who guided me into a project of designing and implementing temporal relationships on top of an entity resolution system. It was a very interesting project, and these experiences enriched my skills and understanding of the field.

This dissertation benefited from many great feedback provided by my thesis com-

mittee members, Susan Davidson, Val Tannen, Andreas Haeberlen, and external member Tyson Condie. I would also like to thank Susan Davidson, Val Tannen and Andreas Haeberlen to sit on my WPE-II committee.

I would like to give my special thanks to Mike Felker for helping me go through many tedious details of administrative tasks, and also to Cheryl Hickey and many other staff at the department who made my life much easier.

My time at Penn would not be so enjoyable without the friends I made at Penn, including all the DB group members: Medha Atre, Olivier Biton, Sarah Cohen-Boulakia, Ling Ding, Anat Eyal, Todd J. Green, Xiaocheng Huang, Marie Jacob, Grigoris Karvounarakis, Svilen Mihaylov, Sudeepa Roy, Julia Stoyanovish, Nicholas Taylor, Allen Zhepeng Yan, Nan Zheng; as well as many Chinese friends with whom I spent many memorable weekends: Ling Ding, Peter Du, Liang Huang, Xiaocheng Huang, Changbin Liu, Qian Liu, Tingting Sha, Gang Song, Wenqing Wang, Bin Yan, Zhepeng Yan, Zhuoyao Zhang, Liming Zhao, Jianzhou Zhao, Wenchao Zhou, Qihui Zhu, and many others.

I would also like to take this opportunity to thank my parents Xindong Liu and Lihua Hu, for their love, encouragement and sacrifice. This dissertation is not possible without their support for me and my family, and I dedicate this dissertation to them.

Last but not least, I dedicate this dissertation to my husband, Zhuowei Bao. We have known each other since high school, and made a decision to come to the United States to pursue our graduate studies together. Throughout the years, we have underwent everything together, and I cannot imagine finishing this dissertation without the support and encouragement from him. This is like a dream come true.

ABSTRACT

INCREMENTAL PROCESSING AND OPTIMIZATION OF UPDATE STREAMS

Mengmeng Liu

Zachary G. Ives and Boon Thau Loo

Over the recent years, we have seen an increasing number of applications in networking, sensor networks, cloud computing, and environmental monitoring, which monitor, plan, control, and make decisions over data streams from multiple sources. We are interested in extending traditional stream processing techniques to meet the new challenges of these applications. Generally, in order to support genuine *continuous* query optimization and processing over data streams, we need to systematically understand how to address *incremental* optimization and processing of update streams for a rich class of queries commonly used in the applications.

Our general thesis is that efficient incremental processing and re-optimization of update streams can be achieved by various *incremental view maintenance* techniques if we cast the problems as incremental view maintenance problems over data streams. We focus on two incremental processing of update streams challenges currently not addressed in existing work on stream query processing: incremental processing of transitive closure queries over data streams, and incremental re-optimization of queries. In addition to addressing these specific challenges, we also develop a working prototype system ASPEN, which serves as an end-to-end stream processing system that has been deployed as the foundation for a case study of our SMARTCIS application. We validate our solutions both analytically and empirically on top of our prototype system ASPEN, over a variety of benchmark workloads such as TPC-H and LinearRoad Benchmarks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decade, with the decreasing cost and the growing adoption of wireless sensors, embedded devices and mobile phones, we are seeing a new class of monitoring and control applications that instrument the environments for security, environmental control, intelligent resource management and human assistance. Examples of such applications include intelligent power grids, smart hospitals, home health monitors, energy-efficient data centers, and building visitor guides. To support these applications, there is a need for systems to bring together disparate data from databases (e.g., site information, patient treatments, maps) with data from the Web (e.g., weather forecasts, calendars), from streaming data sources (e.g., resource consumption within a server), and from devices embedded within an environment (e.g., generator temperature, RFID readings, energy levels). Data has been generated at an unprecedented rate, and how to store, manipulate and make sense of this enormous amount of data becomes a crucial problem. It is increasingly important to be able to turn the raw observations into high-level decision-making and control, such as triggering alarms, identifying problematic users, notifying overloaded machines, or predicting abnormal events.

There are many fundamental questions about how to develop these monitoring and control applications in this emerging sensing world. Motivated from decades of success from relational data management systems (RDBMS), we seek a programming environment and runtime system that can be generalized to any application domain. The most important notion we can learn from relational data management systems is "data inde-

pendence", which encompasses two levels. From the physical level, it means that user applications should be immune from the changes on how to access the data, such as the change to a storage device, or an indexing strategy. From the logical level, it means that higher level abstractions such as user views should be hidden away from the changes to the logical schema, such as tables, columns or rows. These data independence notions can be easily applied to data management of monitoring and control applications as well. In order for our intelligent environments to reach their full potential, what is necessary is an extensible, multi-purpose data acquisition and integration substrate through which the applications can acquire data — without having to be coded with special support for new devices or network types. The key question is how to develop this unified declarative query and integration substrate, which supports a multitude of stream and static data sources on heterogeneous, possibly unreliable networks. Computation should be expressed in a single query language and "pushed" to where it is most appropriate, taking into account capabilities, battery life, rates of change, and network bandwidth.

In the last several years, the database community has started to develop novel solutions for managing these monitoring and control applications: a handful of data stream management systems from both industry and academia have been proposed [14, 24, 51, 77, 88]. These data stream management systems (DSMS) have three major differences compared to traditional relational data management systems (RDBMS). First, since data sources become continuous, DSMSs require a *dynamic* data model, which usually makes "timestamp" first-class citizens. Here the "timestamp" could mean the data generation time, or the data arrival time, that could be associated with each single data tuple. Dynamic data sources also make the data model *evolve*: traditional set semantics of relational schema needs to be replaced by mutable data models. Second, the computational model in DSMSs needs to be *continuous* rather than static. Since data streams keep arriving to the systems, in order to compute results based on a pre-defined query for the newly arrived data, DSMSs need to continuously compute the results in a streaming non-blocking fashion. Finally, DSMSs usually require query planners, or query optimizers, to *continuously* search for the best execution plans based on the evolving streaming data characteristics.

1.1 Motivations

In the following paragraphs, we start with three prominent examples from the emerging monitoring and control applications and discuss their unique challenges and requirements.

Example 1. One of the central questions any major network ISP provider is interested in is, given any two access points in the network, whether they are connected at any given point of time, and if connected, what is the optimal path between any two access points. An access point may join or leave the network, and therefore the paths between the access points are usually dynamic. The need for connectivity monitoring is even more pressing in the realm of sensor networks, where any individual sensor is more likely to run out of power or become unresponsive. All-pairs reachability and shortest paths is usually the core computation behind proximity-type queries, which is extremely valuable to applications that require an understanding of contagiousness and influence of networks. Hence, it is often a challenging and important task to understand how to efficiently compute and maintain the all-pairs end-point connectivity status. □

Example 2. With more and more sensors, RFIDs and embedded devices instrumented in the environments, we have a huge opportunity to integrate the data gathered from the physical world, with the data that we store in databases and data warehouses in static forms, to drive new insights into decision-making and control. To make such integrated applications successful, systems need to bring together data from different sources, static or dynamic, and transform those raw observations through complex logic into smart alerts or decisions. For example, to build a smart hospital, we could instrument the clinics and medical devices with sensors, so that we could monitor the status of patients in real-time.

In these real-time environmental monitoring applications, usually it requires systems to continuously monitor various signals to generate meaningful output in real-time. Decision-making and control tasks can become very complex when they correlate multiple data sources and perform complex operations over them. Hence, finding the best execution strategy can be particularly important: an inferior strategy may be overwhelmingly bad and will impact the timeliness of streaming result delivery. Indeed, there are

plenty of decisions to make on how to execute a complex task, e.g., which two data stream sources to join first, which function to carry out one before another, whether the computation should be performed in-network or at a centralized place, and so on and so forth. As the search space grows larger, and as it is in general hard to estimate how bad a strategy could become, *planning* or *optimizing* for the most cost-efficient strategy could become quite expensive itself. Indeed, as the data characteristics of stream sources become volatile, constant *re-planning* is necessary to meet the up-to-date conditions, since a locally optimum plan is unlikely to become optimal forever and may deteriorate the overall performance of real-time execution. □

Example 3. As cloud computing becomes increasingly popular, more and more companies and institutions need to instrument real-time monitoring, alerting and anomaly detection into their normal 24 * 7 operation of their data centers. For example, a public cloud service provider needs to monitor the user behaviors across multiple machines to determine the aggregate usage to avoid a user from impacting others on a multi-tenancy cluster. Hence, multiple data streams need to be monitored, such as the CPU consumption, memory consumption or network usage for each user at every machine; and then be combined, correlated and aggregated to determine important global indicators such as the energy efficiency of the entire data center.

Real-time monitoring and alerting requires timely data stream integration, execution and optimization. A cost-based optimization engine for data streams allows the system to spend the majority of its resources on query execution once the various cost parameters have been properly calibrated: it can be applied to highly complex plans and has the potential to provide significant benefit if a cost estimation error was made, but it should incur little overhead if a good plan was chosen. Unfortunately with a large number of bursty data sources and complex queries, traditional cost-based query optimization techniques from relational data management systems is too expensive to perform frequently. In order to reduce the overhead, we hope to only modify the parts of computations when necessary, that is, incrementally, across multiple query re-optimizations. It is ideal if we always guarantee the best plan returned by a query re-optimizer, during cost-based adaptivity query processing, but incur little overhead when most of the computations can be

shared with previous rounds of query re-optimizations. Here the goal of a continuous optimizer for data stream management systems is often to optimize the aggregate total of latency of query results, returned in a pipelined real-time fashion, rather than a one-shot optimization performed on the static data. □

Unfortunately there are still several limitations of existing data stream management systems that make them insufficient to fully support monitoring and control applications. In general, in order to support genuine *continuous* query optimization and processing over data streams, we need to systematically understand how to address *incremental* optimization and processing of update streams for a rich class of queries commonly used in the applications. Specifically, there are two main challenges that we found in existing data stream management systems that we aim to systematically address in this dissertation. The first is the incremental processing of transitive closure queries over dynamic streams, and the second is the incremental *re-optimization* of queries. We will illustrate both challenges in more detail below.

Challenge A: lack of support for *incremental* processing of transitive closure queries over dynamic state. As illustrated in our motivating examples, monitoring and control tasks could become fairly *complex*. In particular, *transitive closure* queries are frequently used to measure the reachability and influence in various kinds of dynamic networks. Unfortunately, existing data stream management systems do not have native support for handling recursive queries over dynamic data streams. Most times, they would require application developers to express even a simple recursive query, such as a reachability-style transitive closure query, to be written in a user-defined function. This approach is very flexible. However, it is also error-prone because it requires the application developers to carefully engineer the logic for recursive queries over dynamic streams without assurance of accuracy. More importantly, without recursion supported as first-class citizens in the execution graph, data stream management systems lacks the ability to reason about incremental result generation, in a way that resembles incremental hash joins and incremental aggregates, hence losing the opportunity for generic optimization, parallelism and common tactics to enable efficient incremental computation.

In order to fully support native recursion in data stream management systems, every

core component of the system, from the query language, to the query execution engine, requires major enhancements. At the minimum, the query language needs to have the expressiveness of recursive queries. From the query execution point of view, a fixpoint operator is necessary and a loop needs to be introduced to the common DAG model of the execution graph. To make the problem more challenging, under the data stream model, data keeps arriving to the systems, hence we need to resort to *incremental* recursion processing techniques to enable efficient query result generation. In the presence of both insertions and deletions in update streams, sometimes a single update may have cascading effects on the results of the entire query. Hence, it is important, yet challenging, to design efficient generic incremental approaches to handle recursive queries over dynamic data streams.

Challenge B: incapable of re-using work across multiple query re-optimizations. In addition to handling recursion, the problem of supporting rapid *adaptation* to runtime conditions during query processing is essential as well. A query optimizer for data streams typically relies on the data characteristics to make the correct cost estimations, and the correctness of such cost estimations is vital to the quality of the plan picked by a query optimizer. However, in the dynamic stream scenarios, data characteristics and properties and the availability of system resources may be constantly changing. It is very difficult to effectively choose a good plan for the entire data streams: data statistics may be unavailable or highly variable; cost parameters may change due to resource contentions or machine failures; and indeed a *combination* of query plans might perform better than any single plan. Hence, there is a real need to support rapid *re-optimization*, that is, *continuously* searching for the most cost-efficient plan upon changes of statistics. However, existing DSMSs have no or very limited support for constant query *re-optimizations*. Indeed, in many scenarios, the optimal execution plan would only change when cumulative changes of statistics take into effect. Often times most of the explorations and cost estimations of plans can be reused; and a re-optimizer could only re-explore query plans whose costs are affected by an updated cardinality or a delta cost value. If a re-optimizer does not perform *incremental* computations across rounds, the overall overhead will quickly accumulate and will potentially prevent frequent adaptations from happening because

of delayed latency. We find that the ability to re-use the work across a series of query re-optimizations greatly affects the overall performance of query processing over the lifetime of stream query processing. This is a problem not well-studied in the context of full-fledged cost-based query optimizer (instead of heuristic-based query optimizers) for data streams, because there lacks a systematic analysis of the query re-optimization problem in a full cost-based manner. There are also huge challenges on how to *incrementally* re-optimize while retaining the capabilities of *pruning* effectively.

Interestingly, although these two challenges seem unrelated from the first glance, indeed, they both concern with *incremental* processing of *dynamic state*. In the first challenge, the *dynamic state* refers to dynamic network links, node arrival and departure, and so forth. In the second challenge, changes in data statistics and cost parameters of candidate plans construct another form of *dynamic state*. This second form of state can also be regarded as *metadata* state, e.g., statistics and summaries of data sources. This view of the query re-optimization problem as another instance of the *incremental* computation problem over dynamic state sheds some new insights on how to address these problems in a generic and principled way. In both cases, we need to perform some complex computations over the dynamic state in a continuous fashion. In order to achieve desirable efficiency, we aim to demonstrate the feasibility and benefits of processing a task *incrementally*, rather than from scratch.

1.2 Contributions and Goals

In this dissertation, we study generic approaches towards incremental recursive query answering over dynamic data streams, and incremental query re-optimization of data streams. Our general thesis is that efficient incremental processing and re-optimization of update streams can be achieved by various *incremental view maintenance* techniques if we cast the problems as incremental view maintenance problems over data streams. In particular, we make the following contributions:

- We formulate the problem of computing recursive tasks over dynamic data streams as a classical *incremental view maintenance* problem, which facilitates many generic incremental view maintenance techniques to address the challenges. We develop

a novel, compact *absorption provenance* as an annotation attached to each data item, which enables us to directly detect when view tuples are no longer derivable and should be removed, where the views are defined over a stream of insertions or deletions. We also develop several heuristics to ensure that the absorption provenance annotation, maintained in a Binary Decision Diagram (BDD), remains compact under different topologies. We implement all of the above solutions in our ASPEN prototype system, and experimentally validate the performance under several different scenarios. Results show that we have orders of magnitude savings compared to prior approaches in various settings.

- We explore whether full-fledged cost-based *incremental* techniques for query re-optimization can be developed, where an optimizer would only re-explore query plans whose costs were affected by an updated cardinality or cost value; and whether such incremental techniques could be used to facilitate more efficient adaptivity in dynamic scenarios. We propose a rule-based and declarative approach to query re-optimization. We develop a formulation of query re-optimization as an incremental view maintenance problem, for which we develop novel algorithms like incremental aggregate selection, incremental reference counting and incremental pruning. We implement our solutions in our prototype system, ASPEN, with comprehensive studies of performance against alternative approaches over a diverse set of workloads. Results show that we have an order-of-magnitude performance gains versus non-incremental approaches to query re-optimization.

- We show that our approaches to incremental re-optimization from the declarative perspective can be easily incorporated into traditional *procedural-based* query optimizers (e.g., bottom-up optimizers with dynamic programming and top-down optimizers with branch-and-bounding), without changing their architectures. We study how to design and implement full-fledged cost-based procedural incremental query re-optimization frameworks and present both analytical and empirical results.

- We illustrate the design and implementation of our end-to-end stream query processing prototype system, ASPEN, and demonstrate a campus-motivated smart building application SMARTCIS as a case study.

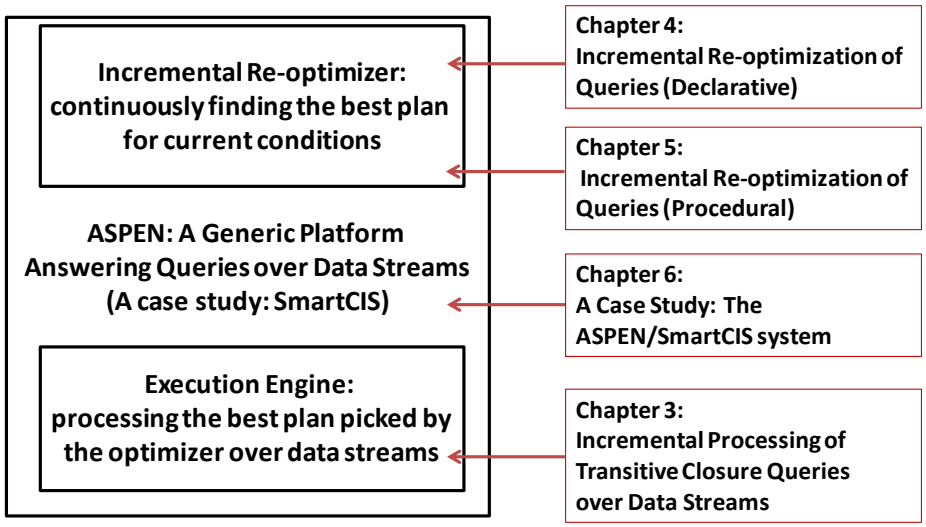


Figure 1: The overview of this dissertation.

Figure 1 shows the roadmap of this dissertation. We will demonstrate incremental *processing* of transitive closure queries over data streams in Chapter 3. In Chapter 4, we will illustrate how to address incremental *re-optimization* of queries through the declarative lens of incremental view maintenance. We then illustrate how our approaches can be easily applied to traditional procedural-based query optimizers in Chapter 5. We will demonstrate the design and implementation of our end-to-end stream query processing prototype system, ASPEN, and demonstrate a campus-motivated smart building application SMARTCIS as a case study in Chapter 6. Below we illustrate the main contributions of each chapter in more detail.

Incremental processing of transitive closure queries over dynamic data streams (Chapter 3).

In this work, we tackle the challenge of efficiently answering transitive closure queries over dynamic data streams. Most of the motivating applications of this dissertation, e.g., networking monitoring and environmental monitoring and control applications, are heavily reliant on *transitive closure* computations. In most of such applications, data streams are generated from distributed devices in a streaming fashion (e.g., from geographically distributed sensor devices instrumented in the environment), and in a lot of scenarios,

centralized monitoring and maintenance is prohibitively expensive in that it requires all the devices to sync the data over the network continuously to be able to maintain the global status. Hence, in these applications, a more realistic assumption is a *decentralized* model, where each node only maintains the local information (e.g., the node's own connectivity to other devices) and can perform some computations on its own. The majority of past work on recursive queries [15, 16] has focused on recursion in the context of centralized deductive databases, and some aspects of that work have ultimately been incorporated into the SQL-99 standard and today's commercial databases. However, recursion is relatively uncommon in traditional database applications, and hence little work has been done to extend this work to a distributed streaming setting. In contrast to prior maintenance strategies for recursive views [40, 45], our work aims to *minimize the overall computation overhead* — both across the network (which is vital to reduce the communication overhead) and inside the query plan (which reduces CPU computational latency). We make the following contributions:

- We specify the connectivity monitoring tasks over dynamic networks as recursive Datalog rules; this enables us to formulate the problem of computing recursive tasks over dynamic data streams as a classical *incremental view maintenance* problem, which facilitates incremental view maintenance techniques to address the challenges.

- We develop a novel, compact *absorption provenance* as an annotation attached to each data item, which enables us to directly detect when view tuples are no longer derivable and should be removed, where the views are defined over a stream of insertions or deletions. This approach improves the overall efficiency (e.g., both network latency and CPU latency) compared to prior approaches [40, 45].

- We develop several heuristics to ensure that the absorption provenance annotation, maintained in a Binary Decision Diagram (BDD), remains compact under different network topologies. This problem of finding the optimal-size provenance is NP-hard; but our heuristics work well in practice.

- We propose a novel *MinShip* operator in the execution engine of a DSMS, which reduces the number of times that tuples annotated with provenance need to be propagated across the network and in the query.

- In order to reduce the propagation of tuples that do not contribute to the answer, we generalize *aggregate selection* [89] to handle data streams. This approach reduces the number of tuples being propagated when aggregations exist.

- We implement all of the above solutions in our ASPEN prototype system, and experimentally validate the performance under several different practical settings (e.g., with different Internet or sensor network topologies). Results show that we have orders of magnitude savings compared to prior approaches in various scenarios.

This work enables a DSMS to efficiently answer transitive closure type of queries over data streams in a potentially decentralized execution engine. It reduces network traffic and query latency compared to previous solutions, hence improves the overall latency of answering queries. This work also provides foundations for a DSMS to be able to enrich its query language to handle recursions (e.g., a query language like Datalog), and it advances the understanding of the general problem of *incremental maintenance* of recursive views in a distributed setting.

Enable incremental query re-optimization in a declarative fashion (Chapter 4).

In data stream management systems, it is increasingly important to support rapid *adaptation* to runtime conditions during query processing. However, it is often not easy to effectively choose a good plan during the course of query execution: data statistics may be unavailable or highly variable; cost parameters may change due to resource contention or machine failures; and in fact a *combination* of query plans might perform better than any single plan.

In the past, query optimization techniques in adaptive query processing systems fall into three general classes: 1) operator-specific techniques that can adapt the order of evaluation for specific combinations of operators rather than the full-fledged space [12, 58]; 2) eddies [10, 33] and related flow heuristics, which are highly adaptive but also continuously devote resources to exploring *all* plans and require fully pipelined execution;

3) approaches that use a cost-based query re-optimizer to re-estimate plan costs and determine whether the system should change plans [54, 56, 72, 85]. Of these, the last cost-based query re-optimizer is the most flexible, in that it can handle all types of expressible queries and not limited to a predefined set of operators or combinations. Perhaps most importantly, a cost-based engine allows the system to spend the majority of its resources on query execution once the various cost parameters have been properly calibrated: it can be applied to highly complex plans and has the potential to provide significant benefit if a cost estimation error was made, but it should incur little overhead if a good plan was chosen. Unfortunately with a large number of bursty data sources and complex queries, standard cost-based query optimizations are too expensive to perform frequently. In order to reduce the overhead, we hope to only modify the parts of computations when necessary, that is, *incrementally*, across multiple query re-optimizations. It would be great if we always guarantee the best plan returned by a query re-optimizer, during cost-based adaptivity query processing, but incur little overhead when most of the computations could be shared with previous rounds of query re-optimizations.

Our goal in this line of work is to explore whether full-fledged cost-based *incremental* techniques for query re-optimization can be developed, where an optimizer would only re-explore query plans whose costs were affected by an updated cardinality or a cost value; and whether such incremental techniques could be used to facilitate more efficient adaptivity in dynamic scenarios.

We address the problem of incremental re-optimization using a novel approach, which is based on the observation that query optimization is essentially a search problem involving the derivation and subsequent pruning of state (namely, alternative plans and their costs). Building an *incremental* re-optimizer requires preservation of some form of state (e.g., , the pruned optimizer memoization table) across optimization runs — but moreover, it must be possible to determine what plans have been *pruned* from this state, and to re-derive such alternatives and test whether they are still viable. Rather than inventing custom semantics for incrementally maintaining state within a query optimizer, we instead adopt an approach of developing an incremental re-optimizer expressed *declaratively*. From a declarative perspective, we can model the ever-changing state in a query optimizer as data, and the query optimization process itself as a task expressed as a *query*

over the data. A query re-optimization problem is essentially an *incremental view maintenance* problem when we model the entire query optimization process as a set of views over the dynamic input data. This also connects back to the previous line of work to support recursive queries over data streams: with an execution engine capable of handle recursions plus some extensions, we can leverage it to execute the query re-optimization task as a query. In essence, we are exploring incrementally processing the results of a query optimizer using a query processor. We develop a variety of novel *incremental* and *recursive* optimization techniques to capture the kinds of pruning used in conventional optimizers, and to generalize them to the incremental case. An incremental optimizer following our model can be competitive with a standard optimizer implementation for *initial* optimization, and significantly faster for *repeated* re-optimizations. Moreover, in contrast to randomized or heuristics-based optimization methods, we still guarantee the discovery of the *best* plan according to the cost model. Specifically, we make the following contributions in this work:

- We define, design and develop a full-fledged cost-based *incremental* query re-optimizer that prunes yet supports incremental re-optimization.

- We propose a rule-based and declarative approach to query re-optimization. We choose an abstraction level of specification that gives us a nice abstraction for incremental processing, effective pruning as well as potential extensions to parallel and distributed architectures. We develop techniques that decouple plan enumerations and cost estimations, relaxing traditional restrictions on search order and pruning.

- We develop a variety of novel logic optimization strategies to prune the computation of query optimization through the new abstraction of the query optimization problem, namely, aggregate selection with tuple source suppression, reference counting and recursive bounding.

- We develop a formulation of query re-optimization as an incremental view maintenance problem, for which we develop novel algorithms like incremental aggregate selection, incremental reference counting and incremental pruning.

- We implement our solutions in our prototype system, ASPEN, with comprehensive studies of performance against alternative approaches over a diverse set of workloads. Results show that we have an order-of-magnitude performance gains versus non-incremental approaches to query re-optimization.

- We demonstrate that incremental re-optimization can be incorporated to good benefit in existing cost-based adaptive query processing techniques [54, 85], over standard LinearRoad [9] benchmarks.

Enable incremental query re-optimization in a procedural fashion (Chapter 5)

In the previous work, we studied how to address the incremental re-optimization problem under the declarative frameworks. In this work, we aim to leverage the insights we gained from the declarative abstractions of the problem, and apply them to the more traditional query optimization frameworks: procedural cost-based query optimization engines. Examples of such engines include bottom-up style query optimizers [47, 86] and top-down style query optimizers [39]. These traditional cost-based procedural-based query optimizers were originally designed for single-pass query optimization, and we strive to understand how to incorporate ideas of incremental re-optimization into these traditional query optimization frameworks. The motivation of this study comes from several observations. First, the traditional query optimization frameworks are widely used in the industry and are already extremely complex, it is usually easier to incorporate the new features in the same framework rather than rewriting everything from scratch. Second, by restricting ourselves to traditional frameworks, we can understand the minimal changes needed to be able to make a procedural query optimizer incremental.

We aim at incremental query re-optimization approaches for the full-fledged cost-based query optimization frameworks, hence our work will be different from other solutions proposed in the past, such as operator-specific techniques (e.g., [12]), eddies-based flow-heuristic techniques (e.g., [10]) or heuristics-based cost-based re-optimization techniques (e.g., [56]). We make the following contributions in this work:

- We define the cost-based procedural incremental query re-optimization problem.

- We leverage the lessons learned from the declarative perspective and present incremental re-optimization algorithms for both bottom-up and top-down style architectures, and discuss optimizations to improve recomputation and book-keeping costs.

- We analyze the worst-case bounds for re-computations of AND and OR nodes during incremental re-optimization.

- We empirically evaluate our procedural incremental re-optimization algorithms and study the performance differences between top-down and bottom-up style incremental re-optimization, and the difference to their non-incremental counterparts. We study the effects of different optimizations, and understand the empirical results of the ratio of recomputed AND and OR nodes during incremental re-optimization.

Aspen: a prototype stream query processing engine (Chapter 6).

We will describe the design and implementation of our own end-to-end stream query processing prototype system, Aspen, and demonstrate a campus-motivated smart building application SmartCIS as a case study.

SmartCIS is motivated by the need to deploy a campus-wide smart building infrastructure for students and faculties. We can instrument our buildings and laboratories with physical sensors, where the status of the room occupancy or labs machine availability can be monitored in real-time. We can build lively in-building maps with different sensors deployed in the hallways, and guide students to the nearest resource available with their desired functionalities. SmartCIS is just one of the many promising applications that could be built on top of the Aspen system. Such a physical sensor-instrumented "smart building" infrastructure has the potential to be applied to many other areas as well, such as energy-efficient data centers and smart hospitals.

Our Aspen prototype system is the core stream query processing engine behind these applications, and we design it with the goal that it should be as generic as traditional database systems such as IBM DB2 and Oracle. It also resembles distributed data stream management systems [8, 14, 24, 51] and sensor query pressing systems [32, 36, 71]. However, the key differentiator is how to develop a unified declarative query processing and

integration platform over a multitude of digital and physical data sources. Computation is expressed in a single query language and is pushed to where it is most appropriate. We also aim to provide a single data access layer for integrating sensor, stream and database data, regardless of origins. As an end-to-end stream query processing system, ASPEN features a declarative query language (similar to StreamSQL) with a user query interface, a compiler (which transforms the declarative query language into internal representations), a heuristics-based federated query optimizer (which divides a query into subqueries that are executed on either on the physical sensor or digital stream side of the query subsystem), two sub-query execution engines (one over the static data sources and one over the stream data sources), a uniform steam acquisition framework for various data sources such as sensors, streams and static data from the databases, and a graphical display interface to demonstrate the query results. Almost all of these components are challenging and most are active open research areas. In this thesis, we will demonstrate 1) the architectural design of a declarative stream and sensor integrated query processing system; 2) how ASPEN enables a uniform stream acquisition framework for various data sources such as physical sensors, digital streams, web data as well as databases; and finally, 3) how ASPEN uses a federated query optimizer that is able to partition a query specified over heterogeneous data sources into a series of subqueries defined over a specific type of data sources.

ASPEN is relevant to the key topics discussed in this dissertation in several ways. First, at the core of its stream query processing engine, there is a *recursive* query processing engine that implements the core ideas of our solutions to the problem of incremental processing of recursive queries. Since ASPEN is designed to be able to execute recursive queries (such as computing shortest paths over physically deployed sensors), our technical contributions there are a key enabler of a general-purpose stream query processing system. Second, our *incremental query re-optimization* solutions are a natural fit for a stream query processing system like ASPEN. In order to achieve the full potential of a cost-based query optimizer over data stream sources, an adaptive and incremental query optimizer is optimal. Third, ASPEN features many other novelties, such as a heuristic-based *federated query optimizer* over stream and sensor subsystems, a uniform data access layer, and graphical stream-result display. These contributions complement the rest of the thesis in a

way that help the readers understand how to design and implement an end-to-end stream query processing system. ASPEN is also built upon stream and sensor query engines as part of other prototypes developed from our collaborators, such as [75, 76, 92].

This dissertation is organized as follows. We first provide the background materials and fundamental concepts behind the thesis in Chapter 2, in topics like query optimization, query execution and Datalog. Then we address the central topic of the thesis: *incremental processing of update streams* in the following three chapters. In Chapter 3, we discuss our approaches to the incremental *execution* of recursive queries over update streams. In Chapter 4, we introduce our approaches to address the incremental query *re-optimization*, in a novel *declarative* fashion. We then apply our techniques into the more traditional procedural-based query optimization framework and discuss our approaches to incremental query *re-optimization* in a procedural fashion in Chapter 5. In Chapter 6, we describe our prototype system built as part of this thesis, ASPEN, and describe SMARTCIS as a case study to demonstrate how we apply the approaches introduced in the thesis to real-life applications. We then discuss the related work in Chapter 7. Finally, in Chapter 8, we summarize the thesis and present several future directions.

Chapter 3 extends our previous papers [66, 67]. Chapter 4 and Chapter 5 provide the full extended analysis and experimental results of our paper [61]. Chapter 6 is based upon the following papers ([62, 64]) which features the design and implementation of our prototype system ASPEN with a case study of SMARTCIS.

Chapter 2

Background

In this chapter, we illustrate several background topics to this dissertation: query optimization, query execution, and Datalog.

2.1 Query Optimization

In this section, we briefly review query optimization of SQL queries, as typically done in traditional database management systems. Query optimization is the main topic in data management systems because a query optimizer, together with a query execution engine, are the two key components of the core of a data management system. Interesting readers can refer to [25] for a survey on this topic.

Suppose we have an example query as follows.

```
select *
from A,B,C
where A.x = B.x and A.y = C.y
```

This query is a three-way join over relations A, B, and C. There are two different join predicates in the "where" clause, and normally two relations are joined first as a binary join, and the result of the first join is then joined with the third relation, also as a binary join. An SQL query over a relational database can be implemented in various ways. Figure 2 shows one possible physical plan to implement this query. An abstracted representation of such an execution plan is a *physical operator tree*. The nodes represent physical operators and the edges represent the data flow among the physical operators.

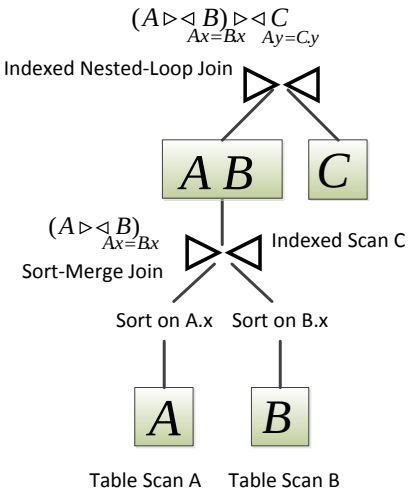


Figure 2: A physical operator tree (plan) for the example query

Each operator takes one or more data streams as input, and outputs a data stream. Examples of physical operators shown on the graph include sort, table scan, index scan, index nested-loop join, and sort-merge join. These physical operators do not necessarily have an one-to-one mapping with their corresponding logical operators (typically joins, selections, projects, aggregates, and unions). For example, "sort" has no corresponding logical operator, hence a logical expression with a specific physical sort order associated with it is usually called an "interesting order". The execution engine is responsible for implementing this physical operator tree that results in generating answers to the query, and in the next section we will review query processing techniques in general.

For complex queries over multiple tables and indexes, there may be millions of different ways to implement a query: each physical operator tree may have different *orders* of combining the tables (yet logically equivalent subject to rewriting rules), different join methods such as hash joins or nested-loop joins, or different interesting orders. Choosing the right method for processing a query over large workloads can produce a query result in seconds, whereas choosing a wrong method can result in queries running for hours, or even days. Query optimization aims at finding the most efficient physical operator tree according to the cost etimations, for the execution engine to process. Query optimization in general can be viewed as a search problem, in particular, it needs:

- A search space of candidate plans (or physical operator trees).
- A cost estimator to assign each candidate plan a cost, which is an estimation of the needed resources to execute a plan, e.g., CPU, I/O, memory, bandwidth, etc.
- An enumeration algorithm to search through the candidate space, and find a query plan with the minimal cost. Optionally, the numeration algorithm can be combined with a pruning mechanism to shrink the size of the search space.

A desirable optimizer should meet the following criteria: 1) the search space includes plans that have the *lowest cost*; 2) the cost estimation is *accurate*, and 3) the enumeration algorithm is *efficient*. It is an nontrivial task to meet these criteria, hence query optimization has been a hot research area since the 80's.

Here we briefly review two different query optimization architectures that are practically successful over the last decades. Specifically, we call them System R-style bottom-up dynamic programming optimizers [47, 86] (System R was prototyped at IBM Research Almaden and later used in IBM DB2 and Oracle), and Volcano-style top-down memoization with branch-and-bound pruning optimizers [38, 39] (used in Microsoft SQL Server). Note that we only review full-fledged cost-based query optimizers here: heuristics for a certain subset of the search space could be applied, but in general transformations do not necessarily reduce costs and therefore must be applied in a cost-based manner by the enumeration algorithm to ensure consideration of low cost plans. Other randomized or progressive query optimization techniques [72] have also been proposed but they are not included in our discussions here.

Bottom-up style optimizers: bottom-up search with dynamic programming [47, 86]. We first review Bottom-up style optimization frameworks that feature bottom-up plan enumeration with dynamic programming. We review the design features of these optimizers with respect to the three angles outlined above: search space, cost estimation, and enumeration/pruning algorithm.

- Search space: In a query optimizer, the original search space of a SPJ query corresponds to *linear* sequence of join operators, e.g., $((A \bowtie B) \bowtie C) \bowtie D$, rather than "bushy" plans, e.g., $(A \bowtie B) \bowtie (C \bowtie D)$, in sacrificing the possibility of losing

cost-efficient plans to trade off the efficiency of query optimization itself. These optimizers utilize a rule-based query rewrite transformation module before carrying out cost-based optimization. Query rewrite translates the original query graph into a semantically equivalent query graph using expansion rules [68], and rewrite rules include, for example, logically equivalent algebraic expressions subject to commutativity or associativity of algebraic operators. Next, logical operators are extended into physical operators to enlarge the search space in order for the system to find the most efficient implementation in an extensible manner. For example, a join operator can use either a nested loop or a sort-merge implementation, and a scan operator can use either an index scan or a sequential table scan. There are also physical operators that may not correspond to any logical operator, e.g., a sort operator is a physical operator. These are often referred to as *physical properties*. Indeed, one of the most important design feature of System R is the consideration of *interesting orders*, that is, the combination of logical operator and physical properties form the characteristics of a candidate plan in the search space.

- Cost estimation: The cost model used by a query optimizer assigns an estimated cost to each partial or complete plan in the search space. Usually a cost model is a linear combination of intermediate result sizes (cardinalities) weighed by factors such as CPU cost, I/O cost, and so forth. Cost models for distributed plans also take into account the communication costs for transferring data. The cost model usually relies on: 1) a set of statistics maintained on relations and indexes, e.g., number of data pages in a relation, number of distinct values in a column, possibly in data structures like histograms; 2) formulas to estimate selectivity of predicates and to project the size of the output data stream for every operator node, for example, the size of the output of a join is estimated by taking the product of the sizes of the two relations and then applying the joint selectivity of all applicable predicates; and 3) formulas to estimate the CPU and I/O costs of query execution for every operator. These formulas take into account the statistical properties of its input data streams, existing access methods over the input data streams, and any available interesting orders on the data stream (e.g., if a data stream is ordered, then the cost

of a sort-merge join on that stream may be significantly reduced). The cost model then uses these statistics and formulas to compute the following information in a bottom-up fashion for operators in a plan: 1) the size of the data stream (cardinality) represented by the output of the operator node; 2) any ordering of tuples created by the output data stream of the operator node; and 3) estimated execution cost for the operator (and the cumulative costs of the partial plan).

- Enumeration/Pruning algorithm: An enumeration algorithm in a bottom-up style query optimizer is analogous to backward-chaining enumeration in the AI literatures. Plans are enumerated *bottom-up* in a stratified way. This stratification starts with enumerating leaf-level access plans, and then enumerates all intermediate plans for combining two of these table access plans, and then three plans, and so forth, until all the plans have been enumerated that produce the overall query results for the original query. These optimizers use *dynamic programming* to prune the search space, based on *the principle of optimality*. That is, in order to obtain an optimal plan for the original query, it suffices to consider only the optimal plans for subexpressions. As the enumeration algorithm is performed in a bottom-up order, we can safely discard suboptimal plans for the subexpressions. The dynamic programming approach reduces the search space from $O(n!)$ to $O(n2^{n-1})$ where n is the number of binary joins in the plan, yet the search problem still has an exponential upper bound.

Top-down style optimizers: top-down search with branch-and-bound pruning [38, 39]. We then review top-down style optimization frameworks (e.g., Volcano, Cascades, Exodus) [39] that feature top-down goal-directed search with branch-and-bound pruning. Note that here we only review high-level designs and their specific differences to the previous bottom-up style optimizers. We discuss them in three angles: search space, cost estimation and enumeration/pruning algorithms.

- Search space: These top-down optimizers universally use *rules* to represent the knowledge of the search space. Two kinds of rules are used [38, 39]: the transformation rules map an algebraic expression into another, and the implementation

rules map an algebraic expression into an operator tree. In comparison to rewriting rules in bottom-up-style optimizers, 1) these systems do not use two distinct optimization phases because all transformations are algebraic and cost-based; 2) the mapping from algebraic to physical operators occurs in a single step; 3) most importantly, instead of applying rules in a *forward chaining* fashion (e.g., in the Starburst query rewrite phase [47]), Volcano-style optimizers perform *goal-driven* application of rules. In regards to linear plans or bushy plans, Volcano-style optimizers consider both flexibly. They consider physical properties as well.

- Cost estimation: Similar to bottom-up style optimizers, cost estimations of top-down style optimizers have to be performed in a bottom-up fashion, because the cumulative cost of the plan depends on the costs of sub-plans and their statistics too.

- Enumeration/Pruning algorithm: Top-down style optimizers achieve dynamic programming in a *top-down* fashion through memoization. When presented with an optimization task, they check whether the query expression has already been accomplished by looking up its logical and physical properties in the table of plans (memoization table) that have been optimized in the past. Otherwise, it applies a logical transformation rule, an implementation rule, or uses an enforcer to modify properties of the data stream. In addition, it uses *branch-and-bounding* to prune the search space, that is, if a sub-plan exceeds the (loose) bound for its representing subexpression (which is normally derived from parents and sibling expressions), it can be safely pruned early and its subtrees do not have to be enumerated. The effectiveness of pruning depends on the top-down search order, and on how fast it reaches the best plan (and hence can use this to prune others effectively). However, pruning is embedded in enumeration and hence has to be in a specific top-down order as well.

Both bottom-up and top-down optimizers achieve the goal of guaranteeing the discovery of the optimal plan with respect to a cost model. The quality of the plan does not depend on which of these search methods is used, but rather on the transformation

rules available for generating plans, as well as the correctness of cost models. Top-down optimizers have an advantage, in that they can prune the search space early with branch-and-bounding, but the effectiveness of pruning depends on the search order, and on how fast it reaches the best plan for a given workload. Hence, a certain sub-plan may be visited many times, although the cost will only be computed once. Bottom-up optimizers prune the search space through dynamic programming (which is pioneered by System R [86]), hence they can discard suboptimal plans, yet they lose the flexibility of branch-and-bound pruning because of backward-chaining application of rules.

2.2 Query Execution

Query execution is the process of actually generating the results of a physical plan chosen by the query optimizer. The goal of query processing/query execution is to compute the plan accurately in the least amount of time. In data stream scenarios, the goal of a query execution engine is often to reduce the *total* latency of delivery of query results (sometimes considering peak or average latency as well); for query execution over distributed machines, the latency of query execution may contain network latency and so forth. In this section, we briefly review standard query execution operators in traditional database systems and their extensions to *pipelined stream-processing* scenarios and *distributed* query execution scenarios.

Standard operators. A static workload processing model is usually a pull-based model, where consumer operators pull the data from producer operators. Generally, a physical operator tree of SQL queries may contain physical operators like scan, join, select, project, function, aggregate, and sort. We discuss each one briefly as follows.

- Scan: A scan operator reads data from the input relation from physical storage. There are many physical scan access methods, such as table scan, indexed scan, and so forth. For example, one can read the entire relation from memory, or build an index and read from a B+ tree or bitmap structure which speeds up retrieval.

- Join: A join operator finds, for each distinct value of the join attribute, the set of tuples in each relation that display that value, and then combine the set of matched

tuples from both relations. A binary join operator could have many physical implementations. The most widely used ones are nested-loop join, sort-merge join and hash join.

A nested-loop join operator essentially joins two relations R and S by making two nested loops as follows: for each tuple r in R, and for each tuple s in S, if r and s satisfy the join condition, then output the tuple (r, s). This requires exactly $|R||S|$ I/O operations, and $|R||S|\sigma$ (where σ is the selectivity of the join predicate) join operations. Advanced approaches to reduce block transfers and disk seeks include enumerating blocks, building an index on the outer relation, and so forth.

A sort-merge join operator first sorts both relations by the join attribute, and then merges the results according to the sort order. Hence, the most expensive part is usually the sorting stage rather than the merging stage.

A hash join operator assumes that the join predicate is an equality predicate (range, comparison or inequality predicates could not use this scheme). There are many hash join implementations, here we briefly present a classical hash inner-join of two relations: first prepare a hash table for the smaller relation in the memory. The hash table entries consist of the join attribute and its row. Because the hash table is accessed by applying a hash function to the join attribute, it will be much quicker to find a given join attribute's rows by using this hash table than by scanning the original relation. Once the hash table is built, it scans the bigger relation and finds the relevant rows from the smaller relation by looking into its hash table. The first phase is usually called the "build" phase, and the second is called the "probe" phase. Finally, the matched tuples are combined.

- Select: A selection operator applies a selection predicate on its input relation. For example, the predicate could be an equality predicate, comparison predicate, inequality predicate and so forth.

- Project: A project operator applies a projection of the input relation on a subset of its attributes. Essentially projection discards certain attributes and retain the rest.

- Function: A function operator applies a function to the input relation. Generally, the functions may contain user-defined functions and built-in functions (e.g., multiplication or subtraction). Sometimes only scalar functions are permitted, e.g., arithmetic or string concatenation.

- Aggregate: An aggregate operator computes the aggregated value of an attribute of tuples subject to the same group-by columns. The operator must know the aggregate column, the group-by columns and the aggregate function. There are many aggregate operator implementations, blocking or non-blocking, hash based or non-hash based. A blocking aggregate operator would read the entire input before computing an aggregate; however, for certain aggregates such as count, sum, max, and min, a non-blocking aggregate is sufficient by allowing the *update* of outputs. A hash aggregate operator essentially computes a hash value of the group-by columns in order to facilitate grouping and merging.

Pipelined stream processing operators. There are two important aspects of extending traditional query processing operators to work in a streaming scenario. First, as data may arrive after the query has been partially executed, in order to produce real-time results, the query execution engine needs to adopt a *pipelined push-based* architecture, i.e., partial results could be returned before receiving more input streams, and results are *pushed* from producers (lower-level operators) to consumers (higher-level operators) in the physical operator tree. Second, as data streams may contain insertions (or arrivals) and deletions (or expirations), every operator needs to extend its data model to support insertions and deletions. Here, we briefly review a popular streaming join operator, the pipelined-hash-join operator, as an extension to its traditional counterpart, the hash-join operator.

A pipelined-hash-join operator is one of the few viable join implementations that is capable of delivering real-time results over stream workloads. Suppose one needs to perform a pipelined-hash-join of relations R and S: $R \bowtie S$. The operator works as follows. Whenever a tuple of R arrives as input, it first stores the tuple in an in-memory hash table, then probes it against any tuples in the hash table of S that match the join key, and then outputs the joined results of the matches. Similarly for S, it does the same

probe. This is a hybrid-hash-join model in that both relations build their hash tables for probing. Usually the hash table of each relation stores the tuples as a map from a hashed value of joined keys to a set of tuples (with the same hashed key), in order to facilitate probing from the other side. The relational algebra of binary joins under set semantics permits pipelining and hash-based schemes, and the only blocking operation in this process is the retrieval and update of hash tables. However, in order to extend it to support insertion/deletion streams, one needs to consider how an insertion/deletion tuple probes against the hash table from the other relation, how to update its own hash table when receiving an insertion/deletion tuple, and how to generate output tuples with the correct semantics of insertion/deletion values. Usually this requires permission of outputting multiple *revision* tuples to amend an initial output result.

Similarly, there exists extensions of traditional operators like functions and aggregates in the streaming scenarios. One needs to ensure that non-blocking semantics is maintained, revisions of outputs are permitted, and insertions and deletions are handled in correct semantics.

Partitioning and distributed operators. In principle, parallel query execution algorithms extend traditional sequential algorithms with multiple threads/cores/machines executing different operations or executing the same operation over partitioned data. Here, we mainly discuss the latter case as the relational algebra of set semantics has the nice property of permitting partitioning of stored data and intermediate results. In this case, every processing machine (or thread) executes the same operation (plan) on horizontally partitioned data, i.e., each machine operates a portion of the input, intermediate results and outputs.

The commonly-used partitioning methods are random, round-robin, range and hash-based partitioning. Random and round-robin partitioning permit perfect load balancing but do not exploit joined properties or duplication elimination and their importance in relational algebra operations such as joins. Range partitioning assigns key ranges (of the partitioning columns) to processing machines. Hash partitioning applies a hash function to determine processing machines. Partitioning of stored data usually determines where initial query operations execute, including selection and projection as well as prelimi-

nary stages of sorting, duplication elimination, and "group by" operations. For binary operations, "co-location" of the two data inputs permits local execution. These binary operations include inner and outer joins, semi-joins with "exists" nested queries, and set operations such as intersection, union and difference. If co-location and local execution is not possible, the input data might need to be *re-partitioned* or *rehashed*. In most cases, both inputs are partitioned based on the joined attributes in the join operator's equality predicate. If one input table is much smaller than the other, it may be more efficient to broadcast the small table to all machines and join the partitioned large table on every machine. The essence of data partitioning strategies for binary joins is to ensure that each pair of input rows that may contribute to the join result "meets" exactly once at a processing machine.

Generally, one can extend a traditional query execution engine to distributed operators by enabling 1) a horizontal data partitioning scheme, 2) partitioning strategies to ensure correct and non-duplicated results, 3) an extension of logical operators to include *re-hashing* or *re-partitioning* operators when co-location and local execution is not possible.

2.3 Datalog

In this section, we review the basic background materials on *Datalog*. Interesting readers may refer to the excellent book [3] for related materials.

An important limitation of relational calculus or relational algebra (and hence basic functionalities of SQL) is that it cannot express queries involving recursion, e.g., paths, transitive closures, etc, natively in the query language. *Datalog* extends conjunctive queries with recursion. A Datalog program consists of a set of rules, each of which is a conjunctive query. Datalog is one of the many languages in *logic programming* and inherits many properties from another popular logic programming language, *Prolog*. One of the most important features of Datalog is its *declarative* nature, compared to the more operational flavor of other programming paradigms, like imperative, object-oriented or functional languages.

Datalog Syntax. Traditionally, the syntax of Datalog follows that of the logic programming language Prolog, with the exception that only constants and relational symbols are

allowed, i.e., there is no function symbol. Recursion is introduced by allowing the same relational symbol in both the heads and the bodies of the rules.

Normally, a Datalog rule has the form:

T(x) :- q(x,y)

Suppose we fix a relational schema T. A Datalog rule has two sides connected by a symbol :-. The left-hand side is called the *head* of the rule, which corresponds to the output of the query; the right-hand side is called the *body* of the rule; and :- conveys the fact that each rule is closely related to a logical implication. T is a relation and q is a conjunction of relational atoms, in this case x and y. $x = x_1, \cdots, x_n$ is a tuple of *distinguished* variables, and $y = y_1, \cdots, y_n$ is a tuple of *existentially quantified* variables. All distinguished variables in the head must appear in at least one atom in the body.

A Datalog program is a finite set of Datalog rules over the same schema. For example, suppose *Link* is a relation representing edges of a graph, the following Datalog program computes the transitive closure of links in the output predicate *Reachable*:

Reachable(x,y) :- Link(x,y)
Reachable(x,y) :- Link(x,z), Reachable(z,y)

Relation symbols (a.k.a. predicates) that appear only in the body of the program's rules are called *edb* predicates (such as *Link*), while those that appear in the head of some rules in the program are called *idb* predicates (such as *Reachable*). A Datalog program defines a Datalog query when one of the idb predicates is specified as the *output*.

Semantics. There are three different yet equivalent approaches to define the semantics of a Datalog program, namely, *model-theoretic*, *proof-theoretic* and *fixpoint* semantics [3].

In the *model-theoretic* semantics of Datalog, each rule is associated with a first-order sentence as follows. First recall that as a conjunctive query, $T(x)$:- $q(x, y)$ corresponds to the first-order query $T \equiv \{x | \exists y q(x, y)\}$. To this end, one can interpret the Datalog rule as the first-order sentence $\forall x(\exists y, q(x, y) \rightarrow T(x))$. Such sentence is a *definite Horn clause* and a Datalog program P may correspond to a set of Horn clauses Σ_P. Let I be an input database instance, in this case an instance of the schema consisting only of edb predicates. A *model* of P is an instance of the entire schema (both edb and idb predicates)

which coincides with I on the edb predicates and which satisfies Σ_p. However, there can be infinitely many instances that satisfy a given program and instance of the edb relations. Thus, logic programming, and consequently Datalog, uses a *minimal* model, i.e., one such that no subset of it is also a model. This is usually understood as a manifestation of the *closed world assumption*: no facts are assumed other than needed. It can be shown that for Datalog, there is exactly one minimal model, which is also the *minimum* model.

In the *proof-theoretic* approach of defining the semantics of Datalog, note that a tuple of constants in a relation can be seen as the head of a rule with an empty body. Such rules are called *facts*. As previously seen, Datalog rules can be associated with first-order sentences. *Facts* correspond to variable-free relational atoms. The main idea of proof-theoretic semantics is that: the answer of a Datalog program consists of the set of facts that can be proven from the edb facts using the rules of the program as *proof rules*. As rule instantiation and application corresponds to standard first-order inference rules, the proof trees are actually rearrangements of the first-order proof. This connects Datalog, through logic programming to automated theorem-proving. One technique for constructing proof is performed in a *top-down* fashion (i.e., starting from the fact to be proven), called SLD resolution. Alternatively, one can start from the base data and apply rules on them to create proof trees for new facts in a *bottom-up* fashion.

The third approach is an operational semantics for Datalog programs stemming from the *fixpoint* theory. The main idea is to use the rules of the Datalog program to define the *immediate consequence* operator, which maps idb instances to idb instances. Interestingly, the immediate consequence operator can be expressed in the SPCU (selection, projection, cross-product, union, but no difference) fragment of the relational algebra, enriched with edb predicates. For example, the immediate consequence operator $\mathcal{F}$ for the transitive closure above is: $\mathcal{F} = \mathit{Link} \bowtie \mathit{Reachable} \cup \mathit{Link}$. One way to think about this operator is that it applies rules on existing facts to get new facts according to the head of those rules. In general, for a recursive Datalog program, the same operator can be repeatedly applied on facts produced by previous applications to it. It is easy to see that the immediate consequence operator is *monotone*. Also it will not introduce any constants beyond those in the edb instances or in the heads of the rules. This means that any idb instance constructed by iterations of the immediate consequence operator is over the active do-

main of the program and the edb instance. This active domain is finite, so there are only finitely many possible idb instances. They are easily seen to form a finite poset ordered by inclusion. Here one of several technical variants of the fixpoint theory can be put to work. The bottom line is that the immediate consequence operator has a *least fixpoint* which is an idb instance and which is the semantics of the program. It can be shown that this idb instance is the same as the one in the minimal model semantics and the one in the proof tree semantics. It can also be shown that this least fixpoint can be reached after finite many iterations of the immediate consequence operator.

Evaluation and Optimization of Datalog. Several techniques have been proposed for the efficient evaluation of Datalog programs. They are usually separated into two classes: *top-down* and *bottom-up* evaluations.

The simplest bottom-up evaluation strategy, also called *naïve* evaluation, is based directly on the fixpoint Datalog semantics. The main idea is to repeatedly apply the immediate consequence operator on results of all previous steps (starting from the base data in the first step) until a step does not yield any new data. It is clear that naïve evaluation involves a lot of redundant computation, since every step recomputes all facts already computed in previous steps. *Seminaïve* evaluation tries to overcome this deficiency, by producing at every step only facts that can be derived using at least one of the new facts produced in the last step (as opposed to all previous steps). In some cases, bottom-up evaluation can produce a lot of "intermediate" tuples that are not used in derivation of any facts in the output relation of the query.

The top-down approach avoids the redundancy problem by using heuristic techniques to focus on relevant facts, i.e., ones that appear in some proof trees of a query answer, especially for Datalog programs with constants appearing in some atoms. The most common approach in this direction is called the query-subquery (QSQ) framework. QSQ generalizes the SLD resolution technique, on which the proof-theoretic semantics are based, by applying it in sets, as opposed to individual tuples, as well as using constants to select only relevant tuples as early as possible. In particular, if an atom of an idb relation appears in the body of a rule with a constant for some attribute, this constant can be pushed to rules producing this idb. Similarly, "sideways information passing"

is used to pass constant binding information between atoms in the body of the same rule. Such constant bindings are expressed using adornments or binding patterns on atoms in the rules, to indicate which attributes are bound to some constant and which are free. *Magic sets* techniques simulate the pushing of constants and selections that happens in top-down evaluation to optimize bottom-up evaluation. In particular, they rewrite the original Datalog program into a new program whose seminaïve bottom-up evaluation produces the same answers as the original one, as well as producing the same intermediate results as the top-down approaches such as QSQ.

Chapter 3

Incremental Processing of Connectivity over Dynamic Networks

As data management systems are handling increasingly distributed and dynamic data, the line between a network and a query processor is blurring. In a plethora of emerging applications, data originates at a variety of nodes and is frequently updated: routing tables in a peer-to-peer overlay network [13] or in a declarative networking system [28, 70], sensors embedded in an environment [32, 71], monitors within clusters at geographically distributed hosting sites [51, 79], data producers in large-scale distributed scientific data integration [40]. It is often natural to express distributed data acquisition, integration, and processing for these settings using declarative queries — and in some cases to compute and incrementally maintain the results of these queries, e.g., in the form of a routing table, an activity log, or a status display.

The queries of interest in this domain are quite different from OLAP/OLTP queries that exemplify centralized DBMS query processing. We consider two main settings.

Declarative networking. In declarative networking [69, 70], an extended variant of datalog has been used to manage the state in routing tables — and thus to control how messages are forwarded through the network. Perhaps the central task in this work is to compute paths available through multi-hop connectivity, based on information in neighboring routers' tables. It has been shown that recursive path queries, used to determine reachability and cost, can express conventional and new network protocols in a declara-

tive way.

Sensor networks. Declarative, database-style query systems have also been shown to be effective in the sensor realm [32, 71], primarily for simple aggregation queries over base streams. Outside the database community, a variety of macro-programming languages [96, 98] have been proposed as alternatives, which include features like regions and paths computations. Declarative languages bring a new abstraction perspective to the problems to enable data independence, however, the query languages and runtime systems must be extended to match the functionality of macro-programming, particularly with respect to computing regions and paths.

We will provide a number of detailed use cases and declarative queries for regions and paths in these two domains. The use cases are heavily reliant on *recursive* computations, which must be performed over distributed data that is being frequently updated in a *stream* fashion (e.g., sensor state and router links are dynamic properties that must be constantly refreshed). In most of such applications, data streams are generated from distributed devices in a streaming fashion (e.g., from geographically distributed sensor devices instrumented in the environment), and in a lot of scenarios, centralized monitoring and maintenance is prohibitively expensive in that it requires all the devices to sync the data over the network continuously to be able to maintain the global status. Hence, in these applications, a more realistic assumption is a decentralized model, where each node only maintains the local information (e.g., the node's own connectivity to other devices) and can perform some computations on its own.

The majority of past work on recursive queries [15, 16] has focused on recursion in the context of centralized deductive databases, and some aspects of that work have ultimately been incorporated into the SQL-99 standard and today's commercial databases. However, recursion is relatively uncommon in traditional database applications, and hence little work has been done to extend this work to a distributed streaming setting. We argue that the advent of declarative querying over networks has made recursion of fundamental interest: it is at the core of the main query abstractions we need in a network, namely regions, reachability, shortest paths, and transitive associations.

To this point, only specializations of recursive queries have been studied in networks.

In the sensor domain, algorithms have been proposed for computing regions and neighborhoods [52, 96, 98], but these are limited to situations in which data comes from physically contiguous devices, and computation is relatively simple. In the declarative networking domain, a semantics has been defined [70] that closely matches router behavior, but it is not formalized, and hence the solution does not generalize. Furthermore, little consideration has been given to the problem of *incremental computation* of results in response to data arrival, expiration, and deletion.

In this chapter, we show how to compute and incrementally maintain recursive views over data streams, in support of networked applications. In contrast to previous maintenance strategies for recursive views [45], our approach emphasizes *minimizing the propagation of state* — both across the network (which is vital to reduce communication overhead) and inside the query plan (which reduces computational cost). We make the following contributions:

- We develop a novel, compact *absorption provenance*, which enables us to directly detect when view tuples are no longer derivable and should be removed.
- We propose a *MinShip* operator that reduces the number of times that tuples annotated with provenance need to be propagated across the network and in the query.
- We develop heuristics to ensure that the absorption provenance structure, maintained in a Binary Decision Diagram (BDD), remains compact.
- We generalize *aggregate selection* to handle streams of insertions and deletions, in order to reduce the propagation of tuples that do not contribute to the answer.
- We evaluate our schemes within a distributed query processor, and experimentally validate their performance in real distributed settings, with realistic Internet topologies and simulated sensor data.

Section 3.1 presents use cases for declarative recursive views. In Section 3.2 we discuss the distributed query processing settings we address. Sections 3.3 through 3.6 discuss our main contributions: absorption provenance, the *MinShip* operator, ensuring compact

provenance, and our extended version of aggregate selection. Finally, we present experimental validation in Section 3.7, and conclude and discuss future work in Section 3.8.

3.1 Distributed Recursive View Use Cases

We motivate our work with several examples that frame network monitoring functionalities as distributed recursive views. This is not intended to be an exhaustive coverage of the possibilities of our techniques, but rather an illustration of the ease with which distributed recursive queries can be used.

Throughout this chapter, we assume a model in which logical relations describe state horizontally partitioned across many nodes, as in declarative networking [69]. In our examples, we shall assume the existence of a relation *link*(*src*,*dst*), which represents all router link state in the network. Such state is partitioned according to some key attribute; unless otherwise specified, we adopt the convention that a relation is partitioned based on the value of its first attribute (*src*), which may (depending on the setting) directly specify an IP address at which the data is located, or a logical address like a DNS name or a key in a content-addressable network [13].

Network reachability. The textbook example of a recursive query is graph transitive closure, which can be used to compute network reachability. Assume the query processor at node X has access to X's routing table. Let a tuple *link*(X,Y) denote the presence of a link between node X and its neighbor Y. Then the following query computes all pairs of nodes that can reach each other.

```
with recursive reachable(src,dst) as
 (  select src,dst
    from link
  union
    select link.src, reachable.dst
    from link, reachable
    where link.dst = reachable.src)
```

The techniques of this chapter are agnostic as to the query language; we could express all queries in datalog, as in [69]. However, since SQL has a more familiar syntax, we present our examples using SQL-99's recursive query syntax. (We assume SQL UNIONs

with set semantics, and that a query executes until it reaches fixpoint. Not all SQL implementations support these features.) The SQL query (view) above takes base data from the *link* table, then recursively joins *link* with its current contents to generate a transitive closure of links. Note that since all tables are originally partitioned based on the *src*, computing the view requires a distributed join that sends *link* tuples to nodes based on their *dest* attributes, who join with *reachable.src*.

There are many potential enhancements to this query, e.g., to compute reachable pairs within a radius, or to find cycles.

Network shortest path. We next consider how to compute the shortest path between each pair of nodes, in terms of the hop count (number of links) between the nodes:

```
with recursive path(src,dst,vec,length) as
 (  select src,dst,src ||'.'|| dst,1 from link
  union
    select link.src,path.dst,link.src ||'.'|| vec,
      length+1
    from link, path where link.dst = path.src)

create view minHops(src,dst,length) as
  (select src,dst,min(length) from path
   group by src,dst)

create view shortestPath(src,dst,vec,length) as
  (select P.src,P.dst,vec,P.length
   from path P, minHops H where P.src = H.src
   and P.dst = H.dst and P.length = H.length)
```

This represents the composition of three views. The *path* recursive view is similar to the previous *reachable* query, with additional computation of the path length, as well as the path itself. The other (non-recursive) views *minHops* and *shortestPath* determine the length of the shortest path, and the set of paths with that length, respectively.

Network highest-bandwidth path. We can similarly define the *highest bandwidth* path: instead of counting the number of links, we instead set a path's bandwidth to be the minimum bandwidth along any link; and then find, for any pair of endpoints, the path with maximum bandwidth.

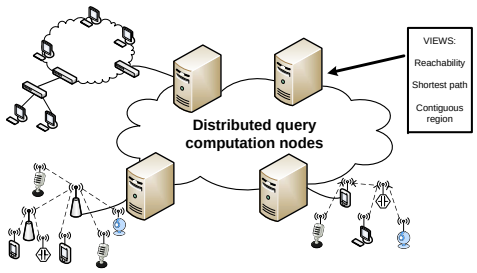


Figure 3: Basic architecture: query processing nodes are placed in a number of sub-networks. Each collects state information about its sub-network, and the nodes share state to compute distributed recursive views such as shortest paths across the network.

Sensing contiguous regions. In addition to querying the graph topology itself, distributed recursive queries can be used to detect regions of neighboring nodes that have correlated activity. One example is a *horizon* query, where a node computes a property of nodes within a bounded number of hops of itself. A second example (which we show and experimentally evaluate in Section 3.7) starts with a series of reference nodes, and computes contiguous regions of triggered sensors near these nodes. This is useful in sensor networks, e.g., in order to determine the average temperature of a fire.

Other example queries. The *routing resilience* query counts the number of paths (alternate routes) between any two nodes. Another class of queries examines multicast or aggregation trees constructed within the network. A query could compute the height of each subtree and store this height at the subtree root. Alternatively, we might query for the *imbalance* in the tree – the difference in height between the lowest and highest leaf node. Finally, a query could identify all the nodes at each level of the tree (referred to as the "same generation" query in the datalog literature).

3.2 Execution Model and Motivations for New Distributed Recursive Techniques

We consider techniques applicable to a variety of networked environments, and we make few assumptions about our execution environment. We assume that our networked query processor executes across a number of distributed nodes in a network; in addition, we allow for the possibility that there exist other legacy nodes that may not run the query

processor (as indicated in Figure 3). In this flexible architecture, the query processing nodes will serve as proxy nodes storing state information (connectivity, sensor status, etc) about devices on their sub-networks: IP routers, overlay nodes, sensors, devices, etc.

Individual sub-networks may have a variety of types of link-layers (wired IP, wireless IP with a single base station, multi-hop wireless/mesh, or tree-structured sensor networks). They may even represent different autonomous systems on the Internet backbone, or different locations within a multi-site organization. Through polling, notifications, or snooping, our distributed query processing nodes can acquire detailed information about these sub-networks. The query processing nodes each maintain a *horizontal partition* of one or more views about the overall network state: cross-sub-network shortest paths, regions that may span physically neighboring sub-networks (e.g., a fire in a multi-story building), etc. During operation, the nodes may exchange state with one another, either 1) to partition state across the nodes according to keys or ranges, or 2) to compute joins or recursive queries.

Importantly, in a volatile environment such as a network, both sensed state and connectivity will frequently change. Hence a major task will be to *maintain* the state of the views, as base data (sensor readings, individual links) are added or deleted, as distributed state ages beyond a *time-to-live* and gets expired, and as the effects of deletions or expirations get propagated to derived data.

3.2.1 Query Execution Model

Our query execution model is a distributed, continuous stream computation, over a set of horizontally partitioned base relations that are updated constantly. We assume that all communication among nodes is carried out using a reliable in-order delivery mechanism. We also assume that our goal is to compute and update *set* relations, not bag relations: we stop computing recursive results when we reach a fixpoint.

In our model, inputs to a query are streams of insertions or deletions over the *base* data. Hence, we process more general *update streams* rather than tuple streams (with annotation of INS/DEL). *Sliding windows*, commonly used in stream processing, can be used to process *soft-state* [82] data, where the time-based window size essentially specifies

the useful lifetime of base tuples. Thus, a base tuple that results from an insertion may receive an associated timeout, after which the tuple gets deleted. When this happens, the derived tuples that depend on the base tuples have to be deleted as well. Due to the needs of network state management, we consider timeouts or windows to be specified over base data only, not derived tuples.

The problem we are trying to solve is incremental view maintenance of recursive queries over update streams. We aim to compute the results efficiently, and optimize the *propagation of state* — both across the network (which is vital to reduce communication overhead) and inside the query plan (which reduces computational cost). The types of recursive queries we consider in this work are 1) linear-recursive queries with at most one idb subgoal in the body of any rule, where no mutual recursion exists; and 2) aggregate queries over linear-recursive views, where the aggregates are the last operators applied. In this regard, the expressive power of the query languages we support in this work is linear-recursive Datalog program, or equivalently, SPJU (Selection-Projection-Join-Union) queries with linear recursion, and with or without aggregates on the top. Most of the recursive queries in our motivating applications fall into these categories (e.g., transitive-closure, reachability, shortest-path queries), and [55] shows how to convert any linear Datalog program into a transitive closure plus small pieces for initialization and for extracting the result. Our goal in this work is thus to address the problem of *incrementally maintaining the views* over update streams, where the views are defined as linear-recursive queries or aggregate queries over them.

3.2.2 Motivation for New Distributed Recursive Techniques

To illustrate the need for our approach, we consider an example. Assume our goal is to maintain, at every node, the set of all nodes reachable from this node. Refer to Figure 4, which shows a network consisting of three nodes and four links (visualized in Figure 5). Each node "knows" its direct neighbors: we represent these in the *link* table, consisting of four entries *link* (*A*,*B*), *link* (*B*,*C*), *link* (*C*,*A*), and *link* (*C*,*B*). As in our previous examples, the *link* table is partitioned such that all values with source *src* are stored on node *src*. In our simple example, there is a direct correspondence between *src* value and location,

reachable(src,dst)
(derivation step 1)

tuple	at → to	pv
(A, B)	A	p_1
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	p_4
(A, C**)**	B → A	$p_1 \wedge p_2$
(B, A**)**	C → B	$p_2 \wedge p_3$
(B, B**)**	C → B	$p_2 \wedge p_4$
(C, B**)**	A → C	$p_1 \wedge p_3$
(C, C**)**	B → C	$p_2 \wedge p_4$

reachable(src,dst)
(derivation step 2)

tuple	at → to	pv
(A, B)	A	p_1
(A, C)	A	$p_1 \wedge p_2$
(B, A)	B	$p_2 \wedge p_3$
(B, B)	B	$p_2 \wedge p_4$
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	$p_4 \vee (p_1 \wedge p_3)$
(C, C)	C	$p_2 \wedge p_4$
(A, A**)**	B → A	$p_1 \wedge p_2 \wedge p_3$
(A, B**)**	B → A	$p_1 \wedge p_2 \wedge p_4$
$*$**(**B, B**)**	C → B	$p_1 \wedge p_2 \wedge p_3$
(B, C**)**	C → B	$p_2 \wedge p_4$
(C, A**)**	B → C	$p_2 \wedge p_3 \wedge p_4$
(C, B**)**	B → C	$p_2 \wedge p_4$
(C, C**)**	A → C	$p_1 \wedge p_2 \wedge p_3$

reachable(src,dst)
(derivation step 3)

tuple	at → to	pv
(A, A)	A	$p_1 \wedge p_2 \wedge p_3$
(A, B)	A	p_1
(A, C)	A	$p_1 \wedge p_2$
(B, A)	B	$p_2 \wedge p_3$
(B, B)	B	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	$p_4 \vee (p_1 \wedge p_3)$
(C, C)	C	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$
$*$**(**A, B**)**	B → A	$p_1 \wedge p_2 \wedge p_3$
$*$**(**B, C**)**	C → B	$p_1 \wedge p_2 \wedge p_3$
(C, A**)**	A → C	$p_1 \wedge p_2 \wedge p_3$
$*$**(**C, B**)**	A → C	$p_1 \wedge p_2 \wedge p_4$

reachable(src,dst)
(derivation step 4)

tuple	at → to	pv
(A, A)	A	$p_1 \wedge p_2 \wedge p_3$
(A, B)	A	p_1
(A, C)	A	$p_1 \wedge p_2$
(B, A)	B	$p_2 \wedge p_3$
(B, B)	B	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$
(B, C)	B	p_2
(C, A)	C	p_3
(C, B)	C	$p_4 \vee (p_1 \wedge p_3)$
(C, C)	C	$(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$

Figure 4: Recursive derivation of *reachable* in recursive steps (bold indicates new derivations). The *at* column shows where the data is produced. The *to* column shows where it is shipped after production (if omitted, the derivation remains at the same node. The *pv* column contains the *absorption provenance* of each tuple (Section 3.3). A tuple marked "*" is an extra derivation only shipped in the absorption provenance model.

although one could decouple each location from its physical encoding by using logical addresses (e.g., doing hash-based partitioning).

Now we define a materialized view *reachable* (*src,dst*), which is also partitioned so tuples with source *src* are stored on node *src*. This query computes the transitive closure over the *link* table, and was shown in the *Network Reachability* example of Section 3.1. Unlike in traditional recursive query execution (e.g., for datalog), here computing the transitive closure requires a good deal of communications traffic: *link* data must be shipped to the node corresponding to its *dst* attribute in order to join with *reachable* tuples (or vice-versa, depending on the query plan); and the output of this join may need to be shipped

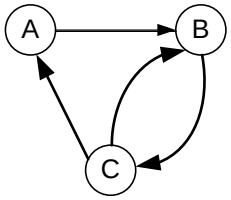


Figure 5: Network represented in *link* relation

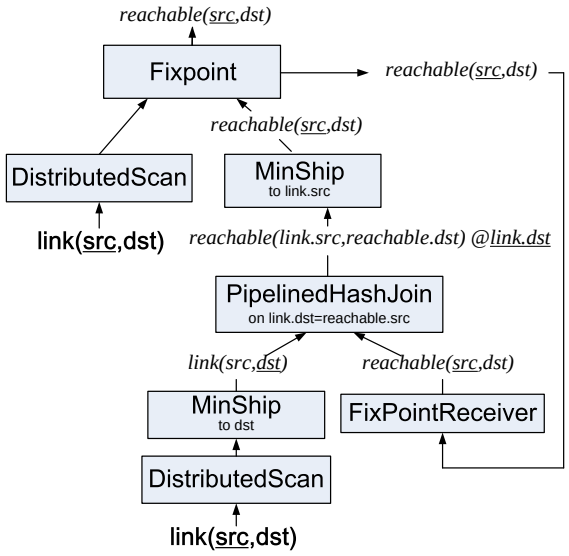


Figure 6: Plan for *reachable* query. Underlined attributes are the ones upon which data is partitioned.

to a new location depending on what its *src* is. Consider the execution plan shown in Figure 6. This plan is disseminated to all nodes, from which it continuously generates and updates partitions of the reachability relation. The left *DistributedScan* represents the table scan required for the base case, which fetches the contents of *link* and sends them to the *Fixpoint* operator. In the recursive case, the *Fixpoint* invokes the right subtree of the query plan: it sends its current contents to a *FixpointReceiver*, where they are joined via a *PipelinedHashJoin* with a copy of *link* — whose contents have been re-partitioned and shipped to the nodes corresponding to the *dst* attribute. The output is shipped to the *Fixpoint* via the *MinShip* (tuple shipping) operator, which in the simplest case simply sends data to a receiving node.

Computing the view instance. Figure 4 steps through the execution of *reachable*, showing state after each computation step in semi-naïve evaluation (equivalent to steps in stratified execution), as well as communication (the "$at \rightarrow to$" columns). We defer discussion of the column marked *pv*.

The base-case contents of *reachable* are computed directly from *link*, as specified in the first "branch" of the view definition (see *Network Reachability* query in Section 3.1). The recursive query block joins all *link* tuples with those currently in *reachable*. Since the tables

are distributed by their first attribute, all *link* tuples must first be shipped to nodes corresponding to their *dst* attribute, where they are joined with *reachable* tuples with matching *src*s. Finally, the resulting *reachable* tuples must be shipped to the nodes corresponding to their *src* attributes. For instance, in step 1, *reachable*(*C*,*B*) is computed by joining *link*(*C*,*A*) and *reachable*(*A*,*B*) as computed from step 0. That requires first shipping *link*(*C*,*A*) to node *A*, performing the join to generate *reachable*(*C*,*B*), and sending the resulting tuple to node *C*. In our figure, we indicate the communication for the resulting *reachable* table in the third column as $A \rightarrow C$.

Since we are following set-semantics execution, duplicate removal will eliminate tuples with identical values; but this only occurs *after* they are created and sent to the appropriate node. For instance, consider *reachable*(*C*,*C*), which is first computed in step 1 and sent to node *C*. During step 2, node *A* re-derives this same tuple; however, it must send this result to node *C* before the duplication can be detected, and the tuple eliminated. In total, 16 tuples (4 initial *link* tuples, and 12 *reachable* tuples) are shipped during the recursive computation. In the final step, a fixpoint is reached when no new tuples are derived. Observe that since we have a fully-connected network, the final resulting *reachable* table at every node contains the set of all node pairs in the network with the first attribute matching the node's address.

Incremental deletion (standard approach). Now consider the case when *link*(*C*,*B*) expires (hence is deleted). Commonly used schemes for maintaining non-recursive views, such as counting tuple derivations, do not apply to this recursive view. Instead, one might employ the standard algorithm for recursive view maintenance, DRed [45]. DRed works by first *over-deleting* tuples conservatively and then *re-deriving* tuples that may have alternative derivations. Figure 7 shows the DRed over-deletion phase (steps 0-4), followed by the rederivation phase (steps 5-8). In the over-deletion phase, it first deletes *reachable*(*C*,*B*) based on the initial deletion of *link*(*C*,*B*). This in turns leads to the deletion of all *reachable* tuples with *src*=*C* (step 1), then those with *src*=*B* (step 2) and *src*=*A* (step 3). The *reachable* table is empty in step 4. DRed will ultimately re-derive *every reachable* tuple, as shown in steps 5-8. Overall, DRed requires shipping a total of 16 tuples, equivalent to computing the entire *reachable* view from scratch, despite having just a single deletion.

reachable(src,dst)
(step 0)

tuple	at
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
(C, A)	C
(C, B)	C
(C, C)	C
$-(C, B)$	C

reachable(src,dst)
(step 1)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
(C, A)	C
(C, C)	C
$-(C, A)$	B → C
$-(C, B)$	B → C
$-(C, C)$	B → C

reachable(src,dst)
(step 2)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
$-(B, A)$	C → B
$-(B, B)$	C → B
$-(B, C)$	C → B

reachable(src,dst)
(step 3)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
$-(A, A)$	B → A
$-(A, B)$	B → A
$-(A, C)$	B → A

reachable(src,dst)
(step 4)

tuple	at → to

reachable(src,dst)
(step 5)

tuple	at → to
(A, B)	A
(B, C)	B
(C, A)	C

reachable(src,dst)
(step 6)

tuple	at → to
(A, B)	A
(B, C)	B
(C, A)	C
(A, C)	B → A
(B, A)	C → B
(C, B)	A → C

reachable(src,dst)
(step 7)

tuple	at → to
(A, B)	A
(A, C)	A
(B, A)	B
(B, C)	B
(C, A)	C
(C, B)	C
(A, A)	B → A
(B, B)	C → B
(C, C)	A → C

reachable(src,dst)
(step 8)

tuple	at → to
(A, A)	A
(A, B)	A
(A, C)	A
(B, A)	B
(B, B)	B
(B, C)	B
(C, A)	C
(C, B)	C
(C, C)	C
(A, B)	B → A
(B, C)	C → B
(C, A)	A → C

Figure 7: DRed algorithm: over-delete and re-derive steps after deletion of *link*(C,B).

In the above example, DRed is prohibitively expensive: deleting a single link resulted in the deletions of all *reachable* tuples; yet, it is clear that nodes A, B, and C are still connected after *link*(C,B) is deleted. One source of deletions in network settings is tuple expirations; a large-scale network tends to be highly dynamic, so tuples will need to expire frequently, thus triggering frequent re-computation and exacerbating the overhead. Perhaps surprisingly, our example illustrates the *common case* behavior for network state queries: most networks are well-connected with bi-directional connectivity along several redundant paths. DRed will over-delete such paths, and then re-derive data.

We have ignored a further issue that DRed must wait until all deletions have been processed before it can start re-deriving. This requires distributed synchronization, which may be expensive.

We now propose a solution that eliminates the need for re-computation, and that also avoids global synchronization. The major challenge with distributed incremental view maintenance lies in handling deletions of tuples. In general, we must either buffer base tuples, then recompute the majority of the query (as in our example); or we must maintain *state* at intermediate nodes, which enables them to propagate the appropriate updates when a base tuple is removed. We adopt the latter approach, developing a scheme that:

- Maintains a concise form of *data provenance* — bookkeeping about the derivations and derivability of tuples — such that it is easy to determine whether a view tuple should be removed when a base tuple is removed. (Section 3.3.)
- Propagates *provenance information* from one node to another only when necessary to ensure correctness — thus reducing network and computation costs. (Section 3.4.)
- Seeks to optimize the encoding of provenance through reordering. (Section 3.5.)
- Propagates *tuples* through distributed aggregate computations only when necessary for correctness — also reducing network and computation costs. (Section 3.6.)

We describe these approaches in the next four sections, with the query plan of Figure 6 as the central example. We then evaluate our methods in Section 3.7.

3.3 Our Approach: Provenance for Efficient Deletions

In order to support view maintenance when a base tuple is deleted, we must be able to test whether a derived tuple is *still derivable*. Rather than over-delete and re-derive (as with DRed), we instead propose to keep around metadata about derivations, i.e., *provenance* [22, 27], also called *lineage* [30].

Provenance alternatives. Different proposed forms of provenance capture different amounts of information. Lineage in [30] encodes the set of tuples from which a view tuple was derived — but this is not sufficiently expressive to distinguish what happens if a base tuple is removed. Alternatives include why-provenance [22], which encodes *sets of source tuples* that produced the answer; and the semiring polynomial provenance representation of [40, 41], whose implementation we term *relative provenance* here. In physical form, the latter encodes a derivation graph capturing which tuples are created as *immediate consequents* of others. The graph can be traversed after a deletion to determine whether a tuple is still derivable from base data [40]. Either of these latter two forms of provenance will allow us to detect whether a view tuple remains derivable after a deletion of a base tuple. However, to our knowledge, why-provenance is always created "on demand" and has no stored representation; and relative provenance relies on the system of equations (encoded

$\sigma_\theta(R)$:	If tuple t in R satisfies θ, annotate t with $\mathbf{P}(t)$
$R_1 \bowtie R_2$:	For each tuple t_1 in R_1 and tuple t_2 in R_2, annotate the output tuple $t_1 \bowtie t_2$ with $\mathbf{P}(t_1) \wedge \mathbf{P}(t_2)$.
$R_1 \cup R_2$:	For each tuple t output by $R_1 \cup R_2$, annotate t with $\mathbf{P}(t_1) \vee \mathbf{P}(t_2)$, where $\mathbf{P}(t_1)$ is **false** iff t does not exist in R_1; similarly for $\mathbf{P}(t_2)$, R_2
$\Pi_A(R)$:	Given tuples $t_1, t_2, \ldots, t_n$ that project to the same tuple t', annotate t' with $\mathbf{P}(t_1) \vee \mathbf{P}(t_2) \vee \cdots \vee \mathbf{P}(t_n)$

Figure 8: Relational algebra rules for composition of provenance expressions. Note that recursive fixpoint incorporates union.

as edges in a graph) to resolve the problem of infinite derivations, which can be expensive in a distributed setting.

Moreover, we note that the tuple derivability problem has several properties for which we can optimize. In particular, base (EDB) tuples may each participate in *many* different derivations — yet the deletion of that base tuple "invalidates" all of these derivations. View maintenance requires testing each view tuple for derivability once base tuples have been removed — which can be determined by testing all of the view tuples' derivations for dependencies on the deleted base tuples.

Our approach: absorption provenance. We define a simplified provenance model, *absorption provenance*, which starts with the following intuition. We annotate every tuple in a view with a Boolean expression: the tuple is in the view iff the expression evaluates to **true**. Let the provenance annotation of a tuple t be denoted $\mathbf{P}(t)$. For base relations, we set $\mathbf{P}(t)$ to a variable whose value is **true** when the tuple is inserted, and reset to **false** when the tuple gets deleted. The relational algebra operators return provenance annotations on their results according to the laws of Figure 8 (this matches the Boolean specialization of provenance described in the theoretical paper [41]).

Our key innovation with respect to provenance is to develop a physical representation in which we can exploit *Boolean absorption* to *minimize the provenance expressions*: absorption is based on the law $a \wedge (a \vee b) \equiv a \vee (a \wedge b) \equiv a$, and it eliminates terms and variables from a Boolean expression that are not necessary to preserve equivalence. We term this model *absorption provenance*. It describes in a minimal way exactly which tuples, in which combinations of join and union, are essential to the existence of a tuple in the

view. The benefit of a compact provenance annotation is reduced network traffic. Even better, we can use absorption provenance to help maintain a view after a base tuple has been deleted: we assign the value **false** to the provenance variable for each deleted base tuple, then substitute this value into all provenance annotations of tuples in the view. If applying absorption to the tuple's provenance results in the value **false**, we remove the tuple. Otherwise, it remains derivable.

Absorption provenance in the example of Figure 4. Absorption provenance adds a bit of overhead to normal query computation: the *Fixpoint* operator must propagate a tuple through to the recursive step whenever it receives *a new derivation* (even of an existing tuple), not simply when it receives a new tuple. Refer back to the *reachable* query example of Figure 4. The *pv* column shows the absorption provenance for every tuple during the initial view computation, with respect to the input *link* tuples annotated p_1, p_2, p_3, and p_4; we see that an additional 4 tuples (beyond the previous set-oriented execution model) are shipped during query evaluation, as a result of computing absorption provenance. For instance, *reachable*(*B*,*B*) is derived in both strata 1 and 2. They have different provenance that cannot be absorbed, hence we must track both derivations.

Absorption provenance shows its value in handling deletions. When *link*(*C*,*B*) is deleted, the only step required with absorption provenance is to zero out p_4 in the provenance expressions of all *reachable* tuples. In this example, zeroing out this derivation only requires two message transmissions, and it does not result in the removal of any tuples from the view. (In the worst case it is still possible that deletions may need to be propagated to all nodes in the network.)

3.3.1 Implementing Absorption Provenance

There are multiple alternatives when attempting to encode an absorption provenance expression. Each expression can, of course, be normalized to a sum-of-products expression, since in the end there are possibly multiple derivations of the same tuple, and each derivation is formed by a conjunctive rule (or a conjunction of tuples that resulted from conjunctive rules). From there we could implement absorption logic that is invoked every time the provenance expression changes. We choose an alternative — and often more compact

in practice — encoding for absorption provenance: the *binary decision diagram* [21] (BDD), a compact encoding of a Boolean expression in a DAG. A BDD (specifically, a *reduced ordered BDD*) represents each Boolean expression in a canonical way, which automatically eliminates redundancy by merging isomorphic subgraphs and removing isomorphic children: this process automatically applies absorption. Since BDDs are frequently used in circuit synthesis applications and formal verification, many highly optimized libraries are available [97]. Such libraries provide abstract BDD types as well as Boolean operators to perform on them: pairs of BDDs can be ANDed or ORed; individual BDDs can be negated; and variables within BDDs can be set or cleared. We exploit such capabilities in our provenance-aware stateful query operators.

Now we describe in detail how to extend traditional database query operators to support absorption provenance of tuples over data streams. Since *Fixpoint* and *Join* are the two main stateful operators in recursive query processing, we discuss how to design provenance-aware *Fixpoint* and *Join* operators.

3.3.2 Fixpoint Operator

The key operator for supporting recursion in database query processing is the *Fixpoint* operator, which combines a *base case* query to produce results, then repeatedly invokes a *recursive case* query. It repeatedly unions together the results of the base case and each recursive step, and terminates when no new results have been derived. The semantics of a provenance-aware *Fixpoint* operator should be: we reach a fixpoint when we can no longer derive any new results that alter the provenance of any tuple in the result.

Unlike traditional semi-naïve evaluation, a distributed *Fixpoint* operator should not block or require computations in synchronous rounds (or iterations), a prohibitively expensive operation in distributed settings. Instead one can use pipelined semi-naïve evaluation [69], where tuples are handled in the order in which they arrive via the network (assuming a FIFO channel), and are only combined with tuples that arrived previously.

In Algorithm 1, we illustrate the pseudocode for a typical provenance-aware *Fixpoint* operator. The *Fixpoint* operator receives insertions from either the base (B^{Δ}) or recursive (R^{Δ}) streams. It maintains a hash table P containing the absorption provenance of each

Algorithm 1 Fixpoint operator

Fixpoint(B^Δ, R^Δ)
Inputs: Input base stream B^Δ, recursive stream R^Δ
Output: Output stream U'^Δ

1: Init hash map P: $U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
2: **if** there is a aggregate selection option **then**
3: Get the grouping key $\overline{uk}$, number of aggregate functions n and aggregate functions $agg_1, \cdots, agg_n$
4: $B'^\Delta := AggSel(B^\Delta, \overline{uk}, n, agg_1, \cdots, agg_n)$
5: $B^\Delta := B'^\Delta$
6: $R'^\Delta := AggSel(R^\Delta, \overline{uk}, n, agg_1, \cdots, agg_n)$
7: $R^\Delta := R'^\Delta$
8: **end if**
9: **while** not $EndOfStream(B^\Delta)$ and not $EndOfStream(R^\Delta)$ **do**
10: Read an update u from B^Δ or R^Δ
11: **if** $u.type = INS$ **then**
12: **if** P does not contain $u.tuple$ **then**
13: $P[u.tuple] := u.pv$
14: Add $u.tuple$ to the view
15: Output u to the next operator
16: **else**
17: $oldPv := P[u.tuple]$
18: $P[u.tuple] = P[u.tuple] \vee u.pv$
19: $deltaPv := P[u.tuple] \wedge \neg oldPv$
20: **if** $oldPv \neq P[u.tuple]$ **then**
21: $u'.tuple := u.tuple$
22: $u'.type := INS$
23: $u'.pv := deltaPv$
24: Output u' to the next operator
25: **end if**
26: **end if**
27: **else if** u is from B^Δ **then**
28: **for** each t in P **do**
29: $oldPv := P[t]$
30: $P[t] = restrict(P[t], \neg u.pv)$
31: **if** $P[t]$ indicates no derivability **then**
32: Remove t from P
33: Remove t from the view
34: **end if**
35: **end for**
36: **end if**
37: **end while**

tuple that it has received, which remains derivable. Note that under our stream data model with provenance annotations, each tuple should include at least three fields: *type*, which indicates whether it is an INS or DEL tuple; *tuple*, which records its raw tuple values; and *pv*, which stores its annotated provenance.

First, if there are aggregate operations on top of the *Fixpoint* operator, we should apply an aggregation operation that might have been "pushed into" the *Fixpoint* — this uses a technique called *aggregate selection* discussed in Section 3.6. Next, upon receipt of

an insertion operation u (Lines 11–26), the *Fixpoint* operator first determines whether the tuple has already been encountered (perhaps with a different provenance). If u is new, it is simply stored in the hash table $P[u.tuple]$ as the first possible derivation; otherwise we merge it with the existing absorption provenance in $P[u.tuple]$. We save the resulting *difference* of provenance in *deltaPv*. If the provenance has indeed changed despite absorption, u gets propagated to the next operator, annotated with the difference provenance *deltaPv*.

Deletions are handled in a straightforward fashion (Lines 27–35), with absorption provenance. Since deletions on the recursive stream are only possibly caused by deletions on the base stream (in our problem setting we only allow delta change originated from base streams), we only need to focus on deletion tuples generated from the base (B^{Δ}) stream. When we receive a deletion operation u, for each tuple t in the table P, we zero out the associated provenance of tuple u ($u.pv$) from the provenance expression of each t ($\mathbf{P}(t)$), computed by BDD operation "restrict" [97] shown in Line 30. If the result is a provenance expression returning **false** (zero), a deletion operation on t is propagated to the next operator after removing its entry from P.

3.3.3 Join Operator

A *PipelinedHashJoin* operator with support for absorption provenance over data streams must modify a conventional pipelined hash join operator in two aspects: 1) provenance-aware state management, and 2) handling insertion and deletion tuples upon arrival.

We show a typical pseudocode of provenance-aware *PipelinedHashJoin* operator in Algorithm 2. This operator needs to maintain in its state similar forms of two pairs of hashtables: suppose when joining relations R and S, the h_R and h_S maintains the tuples indexed on the join keys $\overline{Rk}$ and $\overline{Sk}$ of each R and S tuple respectively, and p_R and p_S maintains the provenance indexed on all attributes of each R and S tuple. The hashtables are similar to those used in the earlier *Fixpoint* operator, except we need to maintain two hashtables, one for each input table.

We will describe in terms of processing a new update tuple u from R^{Δ}. Processing of updates from S^{Δ} is symmetrical. We consider two cases, when u is a deletion or an

insertion. Replacements are treated as a deletion followed by an insertion.

3.3.3.1 Deletions

We first consider the case where u is a delete tuple. We focus on describing the *HalfPipeDel* function in Algorithm 2 invoked by the main *PipeHashJoin* function for each delete tuple u. Two sets of updates need to be performed. First, the internal state (i.e. hashtables of join tuples and respective provenance) maintained by the join operator is updated. Second, new tuples are output from the join, and propagated as deletions up to the next operator according to the query plan.

Operator state update (Lines 1– 8 in *HalfPipeDel*): The provenance state $p_u[u.tuple]$ for tuple u is retrieved, and u's provenance $u.pv$, is deleted from $p_u[u.tuple]$ (In BDD operations, $x - y \equiv x \wedge \neg y$). This essentially clears the absorption provenance of the derivation of u. If $p_u[u.tuple]$ is **false** (zero), which means that all possible derivations of u have been deleted, then we need to remove u from subsequent joins by purging its entry from both p_u and h_u hashtable.

Delete propagation (Lines 9– 16 in *HalfPipeDel*): The deletion of u may cascade the deletion of other tuples when $p_u[u.tuple]$ has been changed. New tuples u' are formed by joining u with matching tuples t in $h_j[u.tuple[\overline{u}k]]$ retrieved using the join key $\overline{u}k$ of u. The absorption provenance of u' is set to the join of $u.pv$ and $p_j[t]$(computed by the BDD as $u.pv \wedge p_j[t]$).The resulting u' tuple with the new absorption provenance is then propagated up the query plan.

3.3.3.2 Insertions

If u is an insert tuple, a similar set of updates are performed. We focus on the *HalfPipeIns* function in Algorithm 2, which is similarly invoked from *PipeHashJoin* for each insert tuple u.

Operator state update (Lines 1– 7 in *HalfPipeIns*): First, the provenance $p_u[u.tuple]$ is retrieved and updated based on the $u.pv$ of the new tuple u. This is required since the new u tuple may have a different derivation from any u that have been derived previously. In addition, if we haven't met u before, we add u to the hashtable h_u indexed by the join

key $\overline{u}k$ of u. Since there may be multiple tuples with a common join key, we maintain a set of them.

Insert propagation (Lines 8– 15 in *HalfPipeIns*): Similar to deletions, the join operator may output new tuples when $p_u[u.tuple]$ has been changed. The h_j hashtable is probed using the join key of u, and each resulting u' tuple from the join is propagated up the query plan. For each matching t in $h_j[u.tuple[\overline{u}k]]$ used in the join, the absorption provenance of u' is set to the join of $u.pv$ and $p_j[t]$ (computed by the BDD as $u.pv \wedge p_j[t]$).

3.3.3.3 Tuple Expirations

To this point, we have discussed the stream join in terms of processing updates *without* considering window semantics. In our algorithm, we encapsulate the window-checking logic in functions W_R and W_S. Each takes a new update, plus the provenance hash table describing the current contents of the relation. As we have previously mentioned, we only support windowing on base relations — for non-base relations, W_R or W_S simply perform no operations and return the empty set. For base relations, the window function 1) updates any relevant internal windowing relations based on the provenance from the new tuple (e.g., advancing the timestamp, or, if the update was an insertion, adding the new tuple's identity as the last-received), and 2) returns the set of tuples that have expired, with the specific provenance terms that have expired.

Calling the window functions (Lines 8–12 in *PipeHashJoin*): . After processing the update, we now pass it along to the window function, so that it may update its internal state. The window function may then return a set of tuples that have expired, with provenance. We delete each tuple from the join.

3.4 Optimizing Propagation of Tuple Provenance

With provenance, each time a given operator receives a new *derivation* of a tuple, it must typically propagate that tuple and derivation, in much the same fashion as it would a completely new tuple. If a tuple is derivable in many ways, it will be processed many times, just as a tuple might be propagated multiple times in a bag relation (versus a

set). This results in undesirable amount of work done in query processing, as well as the amount of state shipped across the network. Even worse, in the general case, a recursive query may produce an infinite number of possible derivations.

Fortunately, absorption helps in the last case. If a new tuple derivation is received whose provenance is not *changed*, we do not need to propagate any information forward. We will reach a fixpoint when we can no longer derive any new tuples that alter the absorption provenance of any tuple in the result. In principle, as networks have finite number of edges, we will reach a fixpoint in finite number of rounds as absorption provenance is a Boolean expression over edge variables.

However, one can take additional steps to reduce the amount of state shipped by distributed query processor nodes. The challenge is to reduce the number of *derivations* (provenance annotations) being propagated through the query plan and the network, while still maintaining the correct semantics to handle deletions. Here one can extend a conventional database *Ship* operator (with no support for provenance) to a stateful *MinShip* operator, in order to maintain provenance annotations about the tuples produced by incoming updates efficiently. This stateful operator can simply propagate the *first* derivation of every tuple it receives, but simply buffers all subsequent derivations of the same tuple by absorbing them into a single Boolean expression. Since we can not control the order of arrival of tuples, this approach is the *best effort* towards keeping record of the tuple with partial provenance. The buffered provenance expression absorbs multiple derivations into a simpler expression, which is more efficient to ship than propagating each one eagerly. This method ensures the eager propagation of tuples but not provenance.

Now if the original tuple derivation is deleted, *MinShip* should propagate forward any alternate derivations it has buffered so far — then it propagates that deletion operation. Additionally, depending on the preferences about state propagation, we can require the *MinShip* operator to propagate all of its buffered state periodically, e.g., when the buffer exceeds a capacity or a time threshold. By changing the batching interval or conditions, we can adjust how many alternate derivations are propagated through the query plan — a smaller interval will propagate more state, and a larger interval will propagate less state. In the extreme case, we can set the interval to infinity, resulting in what we term *lazy provenance propagation*. In the lazy case, alternate derivations of a tuple will only be

propagated when they affect downstream results; this significantly reduces the cost of insertions. (However, in some cases it may slightly increase the cost of deletion propagation.) A typical design of the *MinShip*'s internal state management resembles that of the *Fixopint* operator, and we provide the pseudocode in [65].

3.5 Producing Compact Provenance BDDs

As described previously, encoding and maintaining absorption provenance relies on their compact representation in ordered BDDs. In fact, the compactness of a BDD depends heavily on the order of its variables. Each BDD is a DAG with two terminals, representing 0 and 1. Every internal node in the BDD represents a variable, and every variable appears at a certain level in the DAG, according to a pre-defined ordering of variables. Every internal node has two outgoing edges: one representing the associated variable being assigned **true**, and the other representing **false**. Isomorphic subgraphs in the DAG are merged. All paths leading to the "1" terminal node represent possible truth assignments for the Boolean expression. Some variable orderings lead to different shared subgraphs or to elimination of certain nodes.

Recall that our provenance tokens associated with tuples are converted into BDD variables. We begin by reviewing the BDD variable ordering problem. Then we consider heuristics for ordering the variables as tuple updates arrive during stream processing.

3.5.1 BDD Variable Ordering Problem

Variable ordering has been heavily studied in the BDD literature. Unfortunately, even determining the optimal order of a single BDD is an NP-hard problem [73]. Thus, one must rely on heuristics. Two common heuristics used in practice are *variable-swap* and *sifting* [73]. Variable-swap, as its name implies, seeks to minimize BDD size by trying different swaps of adjacent variables. It is inexpensive but gets trapped in local minima. An extension called sifting searches for a good position for each variable in the order. This is significantly more expensive, but finds better solutions.

Many other techniques have been proposed, including using simulated annealing [19], genetic algorithms [37], or machine learning [23, 42] to guide the search. However, all of

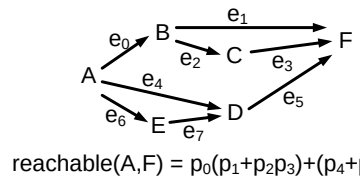
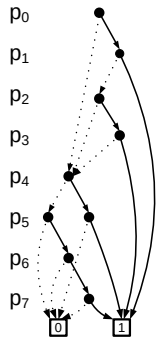
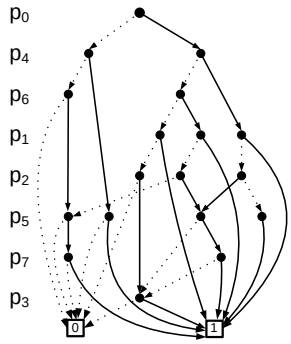


Figure 9: An example network between nodes A and F



(a) Depth-first traversal BDD



(b) Breadth-first traversal BDD

Figure 10: Different variable orders representing the provenance for *reachable*(*A*,*F*) in Figure 9

these approaches assume a setting in which the set of variables and the set of Boolean expressions is known a priori. In probabilistic databases, Olteanu and Huang considered the BDD ordering problem for a certain subclass of queries in [80], but their class is very different from our transitive closure queries.

Our problem does not fall under the standard setting: we are given the task of incrementally computing and maintaining a set of BDD annotations to a set of tuples in a streaming transitive closure computation. We are limited in our knowledge of the Boolean expressions to be merged, as the expressions are formed through evaluating transitive closure queries. Our problem is to incrementally re-order BDD variables (corresponding to provenance tokens) every time we receive and process changes to network link data.

3.5.2 Motivation for Depth-first Traversal Heuristic

Our approach will be to order BDD variables according to depth-first traversal order of the network. To explain the intuition for why, Figure 9 shows an example network with six nodes and eight links (which are unidirectional for simplicity). If we form the BDD

for the connections between start and end, *reachable*(*A*,*F*), following the rules of Figure 8, then we get a provenance expression: $p_0(p_1 + p_2p_3) + (p_4 + p_6p_7)p_5$, where for every $0 \leq i \leq 7$, p_i is the provenance token for e_i.

A depth-first traversal of the graph might visit the nodes in the order e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7. Figure 10(a) shows that this BDD is quite compact, with 9 nodes. A breadth-first traversal might be e_0, e_4, e_6, e_1, e_2, e_5, e_7, e_3. Here, we get an 18-node BDD, shown in Figure 10(b). Note this has twice as many nodes as the previous example.

Intuitively, the BDD is most effective at merging Boolean terms that share initial variables (i.e., segments close to the start node), and a depth-first traversal computes paths in a way that maximizes sharing of these initial variables (segments).

3.5.3 Incremental Depth-first Labeling Algorithm

The previous section gave a rationale for our basic heuristic of extending a BDD in depth-first traversal order. We now describe how to incrementally maintain, as the network graph is being traversed, a global ordering on all variables, such that each BDD will be generated in a fashion that follows a depth-first ordering on the variables. In a distribute setting, this variable ordering process requires global coordination, either through a single central server or through state replication on all nodes. We assume that as new base tuples (*link* for the *reachable* query) are incrementally received by the system, they are fed into a variable reordering algorithm (Algorithm 3 shows the insertion portion; we omit the deletion processing steps but sketch them below). This algorithm incrementally maintains an *edges* vector that establishes an ordering on the network edges (*link* tuples) that conforms to a depth-first traversal.

Given the current state of this vector, we can assign each edge *edges[i]* in the vector to the variable v_i in the BDD, located at depth i. Now, when a union sub-operation (within a *Fixpoint* or *Aggregate*) or join operation occurs, the BDDs associated with the input tuples will conform to the same variable ordering and will combine in (ideally) a compact fashion. As the *edges* vector gets updated, we may need to do fairly inexpensive variable swapping within each BDD (a functionality already supported).

The intuition of the algorithm is that we maintain the *edges* vector in a way that exactly

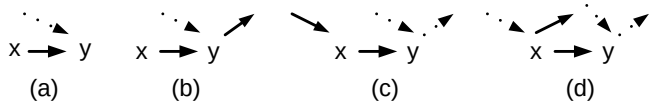


Figure 11: Four different cases when *edge(x,y)* is added. A solid arc represents at least one edge and a dashed arc represents zero or more edges

describes a depth-first traversal of the graph (this includes all cyclic edges). Suppose we have two nodes n and n', which are siblings in terms of the DFS traversal. By definition, we will traverse all edges reachable from node n before those reachable from n'. Hence, if the first edge originating from n is recorded at position $startInx[n]$ in the *edges* vector, then *all* of the edges reachable from n will appear in *edges* before the index position $startInx[n']$. We record the vector position of the *last* edge reachable from n as $endInx[n]$. At initialization, we set all elements of $startInx$ and $endInx$ to the null indicator -1.

Now, to incrementally maintain *edges* and the index positions, we must consider the different scenarios for how some new edge (x,y) may relate to existing paths in the graph. Figure 11 illustrates these cases. Refer to Algorithm 3 for the code describing these scenarios. In case (a) (lines 5-6), x is a new node and y has no outgoing edges. We append (x,y) to vector *edges*. For case (b) (lines 8-14), x is a new node and y has outgoing edge(s). If there exists an incoming edge to y, we append (x,y) to the end of *edges*; otherwise, we insert (x,y) into *edges* before position $startInx[y]$. Case (c) (Lines 17-22) is where x has incoming edges and y is now traversed earlier. We insert (x,y) into *edges* after position $endInx[x]$ and set this index to *startInx[x]*. If $endInx[x] < startInx[y]$, then we move edges in the range $[startInx[y], endInx[y]]$ directly after $endInx[x]$. Finally, case (d) (line 25) is when x already has outgoing edge(s). This is similar to case (c), except we do not modify $startInx[x]$. Note that we choose to insert the new edge to the end of vector *edges* in case (a) and (b), since it does not affect the previous ordering and is the most cost-efficient way to maintain the vector. Incremental insertion requires $O(|edges|)$ operations.

Deletion follows similar principles, but is somewhat more complex. If an edge is removed, this may "orphan" a portion of the original DFS traversal subgraph. We must now scan forward in the *edges* vector to find the *first* edge that references any node n in this orphaned subgraph. Immediately after this edge connecting to n, we insert the edges

(in DFS order) reachable from n. We repeat the process for any remaining nodes from the orphan subgraph, and drop any edges that are no longer connected. If we use hash sets to match nodes in the orphaned subgraph, deletion can be done in $O(|edges|)$ operations.

3.6 Optimizing Propagation of State

Our next challenge is to optimize the amount of state (in terms of unique tuples, not just alternate derivations of the same tuple) that gets propagated from one node to another, while still maintaining the correct semantics of query results over both insertions and deletions. Given that aggregation is commonplace in network-based queries, we need a way to also suppress tuples that have no bearing on the output aggregate values. One can have many alternative ways of reducing the network traffic of aggregate queries, e.g., compression. However, some are orthogonal to what we propose here (e.g., one can apply compression on top of our scheme). We consider a general aggregate optimization technique, *aggregate selection* [89], that pushes the comparison down to the operators before the aggregation (e.g., in order to compute MIN, any result bigger than the cached optimal min should not be propagated), yet still maintaining absorption provenance for the correct semantics of tuple provenance.

Specifically, we adapt *aggregate selection* [89] to a streaming model, with a *windowed* aggregation (group-by) operation [77]. We consider **MIN**, **MAX**, **COUNT**, and **SUM** aggregate functions (**AVERAGE** can be derived from **SUM** and **COUNT**, as in [26].). In essence, the aggregate computation is split between a partial-aggregate operation that is used internally by stateful operators like the *Fixpoint* and *MinShip* to prune irrelevant state, and a final aggregation computation is done at the end over the partial aggregates' outputs. Our main contributions are to support revision (particularly deletion) of results within a windowed aggregation model, and to combine aggregate selection with minimal provenance shipping.

One can achieve the above design principles with an implementation as follows. Our aggregate selection (*AggSel* for short) module, shown in Algorithm 4, can be embedded within any operator that creates and ships state. (Both *Fixpoint* and *MinShip* are able to use this module.) The module takes as input a stream U^{Δ}, a grouping key $\overline{uk}$, the

number of aggregate functions n, and a set of aggregate functions $agg_1, \cdots, agg_n$. The module maintains a hash table H indexed on the grouping key $\overline{uk}$, which records all the buffered tuples met so far based on its grouping key values — this is necessary to support tuple deletion. A corresponding hash table P maps from each tuple to their absorption provenance. Another hash table B is maintained to record the value associated with each aggregate attribute agg_i, for the grouping key $\overline{uk}$. *AggSel* finally outputs a stream U'^{Δ} of the update tuples.

Each time *AggSel* receives a stream insertion (Lines 6–25), it inserts this tuple into the internal map H from group-by key $\overline{uk}$ to source tuple set. (If a tuple with the same value already exists in the set, then it simply updates the provenance P for the tuple.) Next, if the insertion affects the result of any aggregate attribute associated with $\overline{uk}$ — it changes the **MIN** or **MAX** value, or it revises the **AVERAGE** or **SUM** — the aggregation selection module will then propagate a *deletion* operation on the old aggregate value. After checking all the aggregate functions, if at least one of the aggregate values is affected, then it propagates this input insertion tuple as an *insertion*; if none of them is affected, it propagates nothing (see the loop starting at Line 12). Meanwhile, the module applies the change to its internal state.

Upon encountering a stream deletion or an expiration (Lines 25–49), *AggSel* checks whether the deletion has any effect on the derivability of the deleted tuple (Lines 26–28), and then whether any aggregate value associated with the group-by key $\overline{uk}$ is affected. If an aggregate value is modified (i.e., this deletion tuple at least partly determines the aggregate value), then *AggSel* traverses through the current version of buffered tuple table, computes the updated aggregate value, and propagates an *insertion* of the tuple with the new aggregate value. If any of the aggregate values is affected, then it propagates a *deletion*. Meanwhile, the module applies the change to its internal state.

3.7 Experimental Results

We have developed a Java-based distributed query processor over the ASPEN prototype system ([64, 92]) that implements all operators as described in Sections 3.3-3.6. Our implementation utilizes the FreePastry 2.0_03 [84] DHT for data distribution, and JavaBDD

v1.0b2 [97] as the BDD library for absorption provenance maintenance. Experiments are carried out on two clusters: a 16-node cluster consisting of quad-core Intel Xeon 2.4GHz PCs with 4GB RAM running Linux 2.6.23, and an 8-node cluster consisting of dual-core Pentium D 2.8GHz PCs with 2GB RAM running Linux 2.6.20. The machines are internally connected within each cluster via a high-speed Gigabit network, and the clusters are interconnected via a 100Mbps network shared with the rest of campus traffic. Our default setting involves 12 nodes from the first cluster; when we scale up, we first use all 16 nodes from this cluster, then add 8 more nodes from the second cluster to reach 24 nodes. All experimental results are averaged across 10 runs with 95% confidence intervals included.

3.7.1 Experimental Setup

We studied two query workloads taken from our use cases:

Workload 1: declarative networks. Our query workloads consist of the *reachable query* and the *shortest-path* query (Section 3.1). As the input to these queries, we use simulated Internet topologies generated by GT-ITM [43], a package that is widely used for this purpose. By default we use GT-ITM to create "transit-stub" topologies consisting of eight nodes per stub, three stubs per transit node, and four nodes per transit domain. In this setup, there are 100 nodes in the network, and approximately 200 bidirectional links (hence 400 *link* tuples in our case). Each input *link* tuple contains *src* and *dst* attributes, as well as an additional *latency* cost attribute. Latencies between transit nodes are set to 50 ms, the latency between a transit and a stub node is 10 ms, and the latency between any two nodes in the same stub is 2 ms. To emulate network connectivity changes, we add and delete *link* tuples during query execution.

Workload 2: sensor networks. Our second workload consists of region-based sensor queries executed over a simulated 100m by 100m grid of sensors, where the sensors report data to their local query processing node. We include 5 "seed" groups, each initialized to contain a single device. Our recursive view *activeRegion* finds contiguous (within k meters, where by default k=20) triggered nodes and adds them to the group — or removes them

if they are no longer triggered. Based on that, we can compute the the largest such active region.

```
with recursive activeRegion(regionid,sensorid) as
  ( select M.regionid, S.sensorid
    from sensor S, coordSensor M, isTriggered T
    where M.sensorid = S.sensorid
          and S.sensorid = T.sensorid
   union
    select A.regionid, S2.sensorid
    from sensor S1, sensor S2, activeRegion A,
         isTriggered T
    where distance(S1.coord, S2.coord) < k
          and S1.sensorid = A.sensorid and
          S1.sensorid = T.sensorid )

create view regionSizes(regionid,size) as
  (select regionid, count(sensorid)
   from activeRegion
   group by regionid)

create view largestRegion(size) as
   (select max(size) from regionSizes)

create view largestRegions(regionid) as
   (select R.regionid
    from regionSizes R, largestRegion L
    where R.size = L.size)
```

Initially all the seed sensors are triggered. Also we trigger half of the sensors in the network to study the effects of insertions, and then randomly remove them to study the effects of deletions. Note that while the input topology simulates a grid-based sensor topology, the queries are executed over our real distributed query processor implementation.

Our evaluation metrics are as follows:

- **P**er-tuple provenance overhead (B): the space taken by the provenance annotations per-tuple.
- **C**ommunication overhead (MB): the total size of communication messages processed by each distributed node for executing a distributed query to completion.

- **P**er-node state within operators (MB): the total state overhead maintained inside operators on each distributed node.

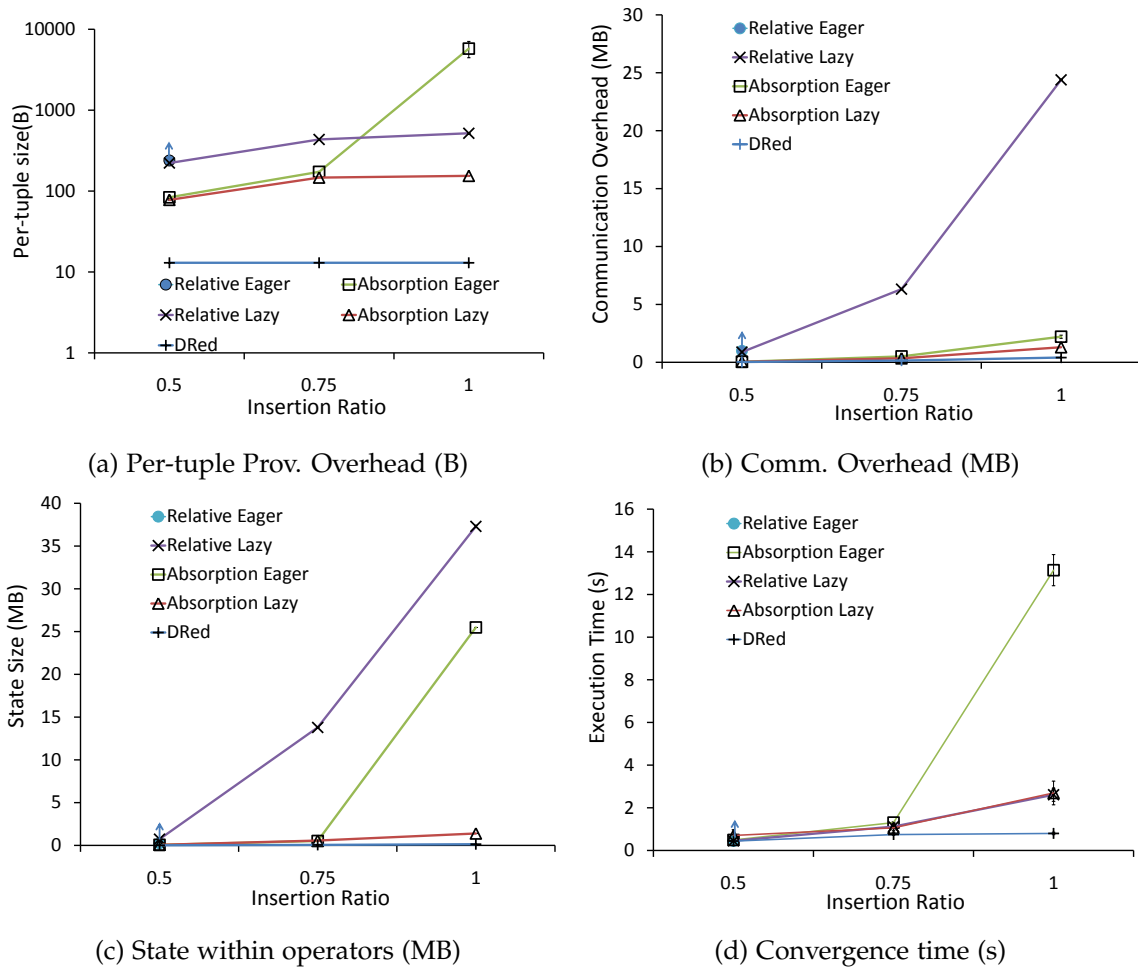
- **C**onvergence time (s): the time taken for a distributed query to finish execution on all distributed nodes.

3.7.2 Incremental View Maintenance with Provenance

Our first set of experiments focuses on measuring the overhead of incremental view maintenance. Using the *reachable* query as a starting point, we compare three different schemes: the traditional *DRed* recursive view maintenance strategy, *relative provenance* [40] where each tuple is annotated with information describing derivation "edges" from other tuples, and our proposed *absorption provenance*. We also consider two schemes for propagating provenance: an *eager* strategy (propagate state from *MinShip* once a second) and a *lazy* one (propagate state only when necessary).

Insertions-only workload. We first measure the overhead of maintaining provenance, versus normal set-oriented execution. Figure 12 shows the performance of the *reachable* query, where the Y-axis shows our four evaluation metrics, and the X-axis shows the fraction of links inserted, in an incremental fashion, up to the maximum of 400 link tuples required to create the 100-node GT-ITM topology. Given an insertion-only workload, *DRed* has the best overall performance, since no provenance needs to be computed or maintained. Relative provenance encodes more information than absorption provenance, resulting in larger tuple annotations, more communication, and more operator state. Relative provenance with eager propagation (*Relative Eager*) did not converge within 5 minutes for insertion ratios of 0.75 or higher; hence, we only show lazy propagation (*Relative Lazy*) for the remaining graphs. Eager propagation with absorption provenance (*Absorption Eager*) also is costly due to the overhead of sending every new derivation of a tuple. Lazy propagation of absorption provenance (*Absorption Lazy*) is clearly the most efficient of the provenance schemes.

Insertions-followed-by-deletions workload. Our next set of experiments separately measures the overhead of deletions: here provenance becomes useful, whereas in the insertion



(a) Per-tuple Prov. Overhead (B)

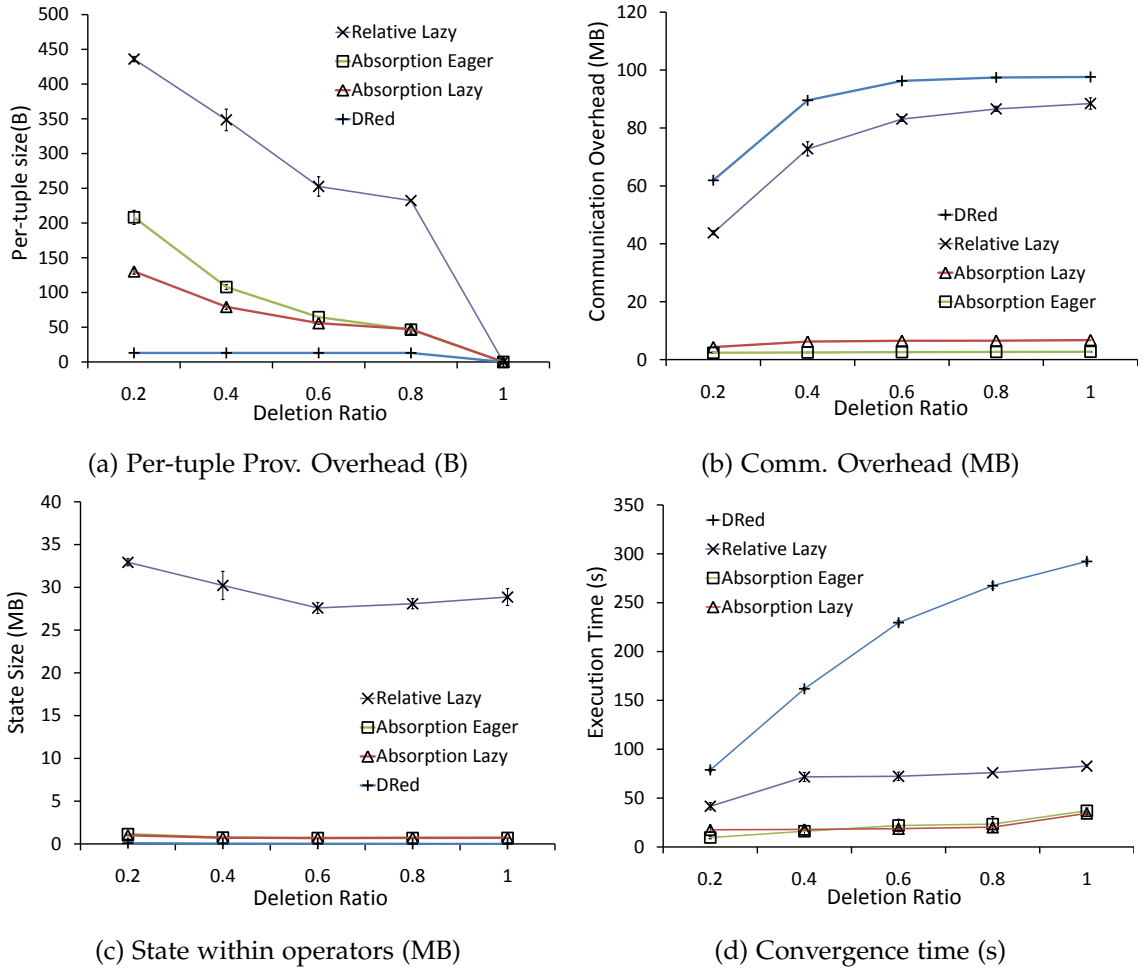
(b) Comm. Overhead (MB)

(c) State within operators (MB)

(d) Convergence time (s)

Figure 12: *reachable* query computation as *insertions* are performed

case it was merely an overhead. (One can estimate the performance over a mixed workload by considering the relative distribution of insertions versus deletions and looking at the overheads on each component.) Given the same 100-node topology, after inserting all the *link* tuples as above, we then delete *link* tuples in sequence. Each deletion occurs in isolation and we measure the time the query results take to converge after every deletion is injected. Figure 13 shows that *DRed* is prohibitively expensive for deletions when compared to our absorption provenance schemes: it is an order of magnitude more expensive in both communication overhead and execution time. Relative provenance wins versus *DRed* in communication cost and convergence time because it does not over-delete and re-derive. However, its performance is worse than absorption provenance, and it



(a) Per-tuple Prov. Overhead (B)

(b) Comm. Overhead (MB)

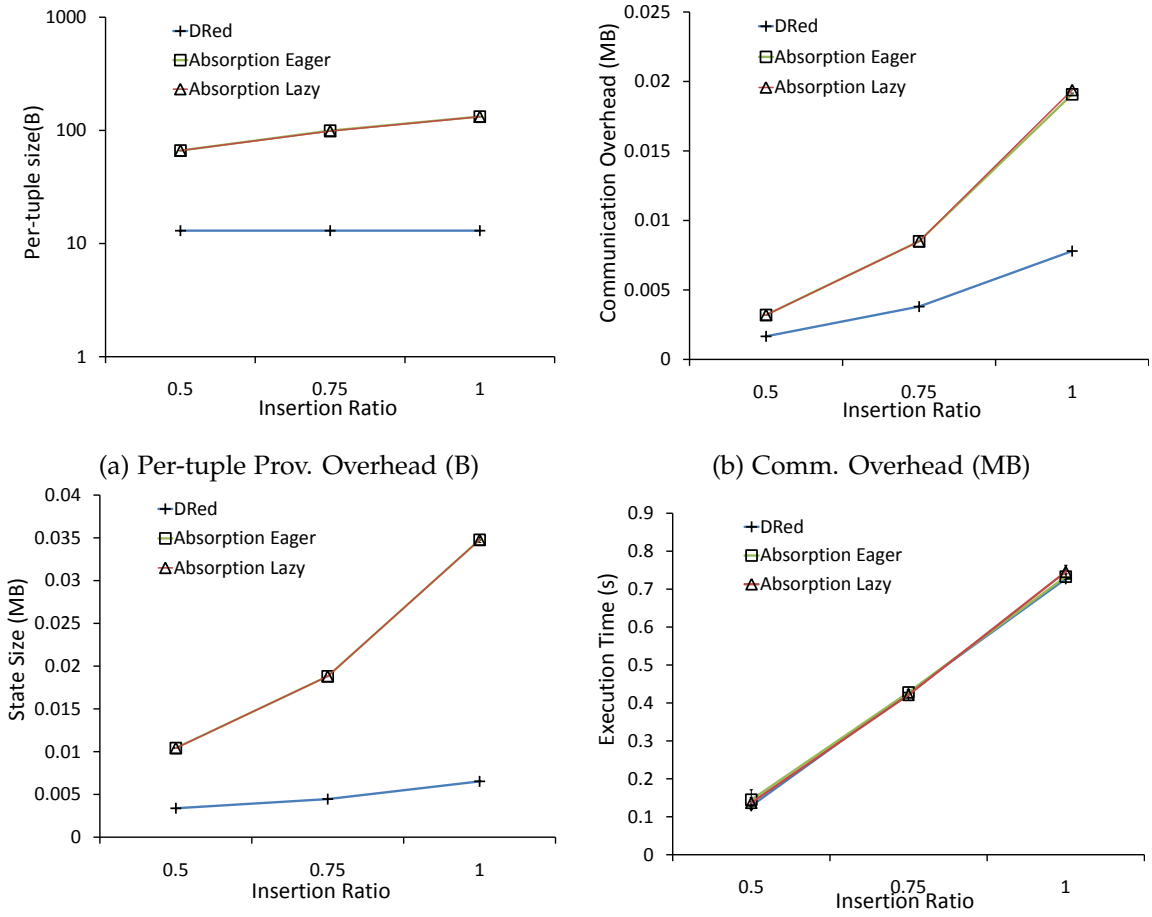
(c) State within operators (MB)

(d) Convergence time (s)

Figure 13: *reachable* query computation as *deletions* are performed

also incurs more per-tuple overhead and operator state. Relative provenance relies on graph traversal operations to determine derivability from base tuples (see [40]), and thus is expensive in a distributed setting. In contrast, absorption provenance directly encodes whether a derived tuple is dependent on a base tuple. Overall, absorption provenance is the most efficient method in deletion handling, and consequently ships fewer tuples than the other methods. Taking both insertions and deletions into account, *Absorption Lazy* has the best mix of performance.

Region-based sensor query. The *region* query is computed over a different topology from the *reachable* case, and it exhibits slightly different update characteristics. Still, as we see in Figure 14, which measures performance with the insertion workload described earlier



(a) Per-tuple Prov. Overhead (B)

(b) Comm. Overhead (MB)

(c) State within operators (MB)

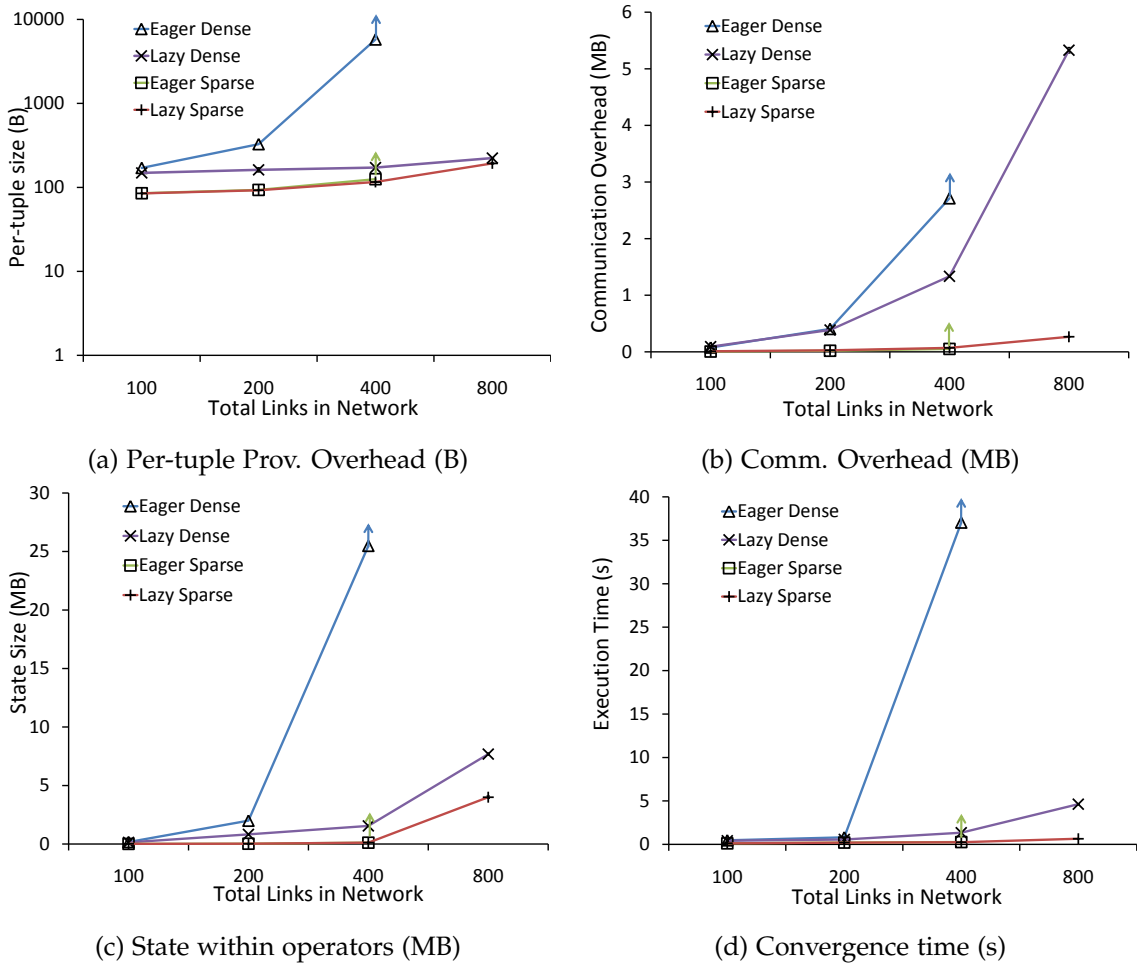
(d) Convergence time (s)

Figure 14: *region* query computation as *insertions* are performed

in the experimental setup, performance follows similar patterns. (The overhead is lower across each of the four metrics, since the network is smaller here and neighbors are within closer proximity.) Under deletion workloads, the trends shown by the *region* query also closely mirror that of the reachable query and those graphs are shown in [65]. Since the queries exhibit similar performance, we focus on the *reachable* query for our remaining experiments.

3.7.3 Scalability

Next we consider how our absorption provenance schemes scale, with respect to inputs and to the query processing nodes.



(a) Per-tuple Prov. Overhead (B)

(b) Comm. Overhead (MB)

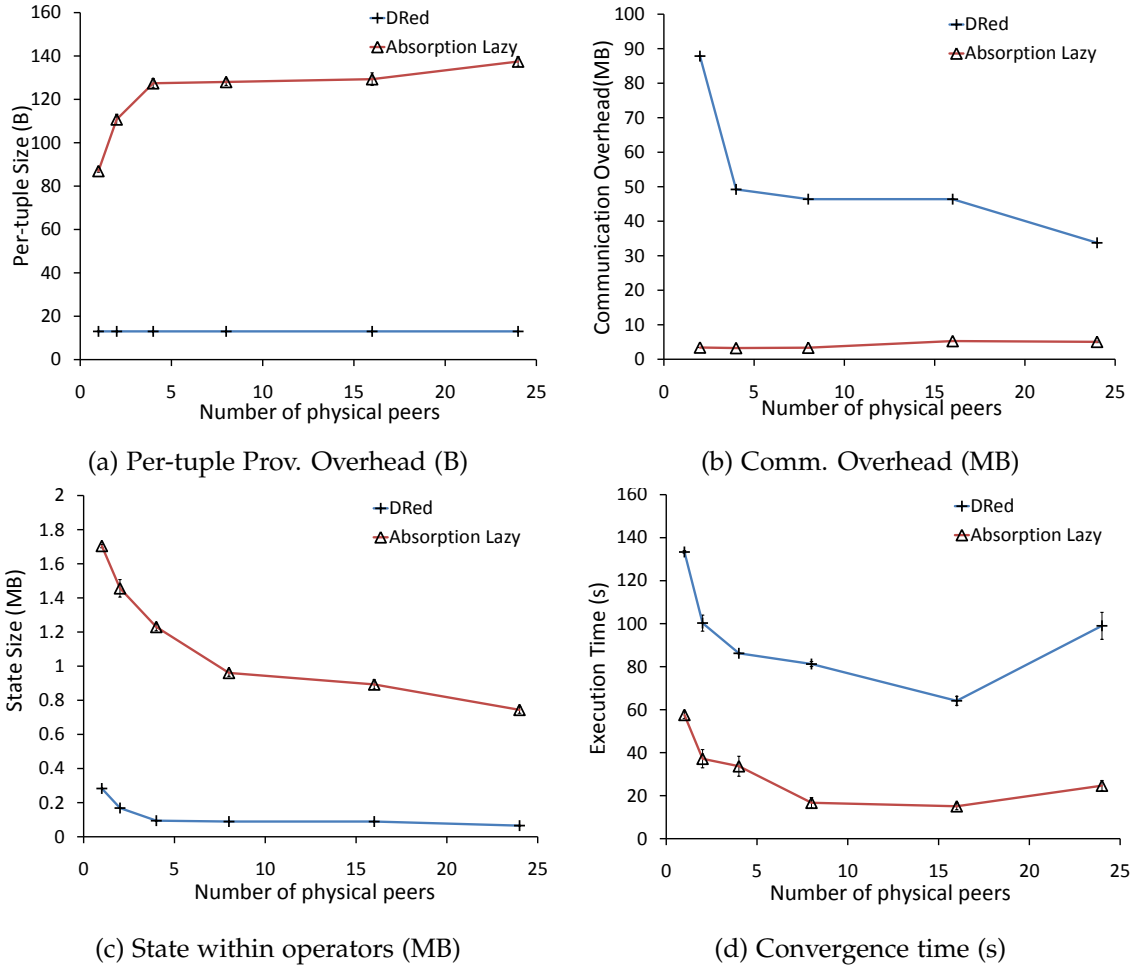
(c) State within operators (MB)

(d) Convergence time (s)

Figure 15: Increasing the number of links (and nodes) for the *reachable* query over inserts

Scaling data. We increase the number of input link tuples, by increasing the average number of transit nodes in the GT-ITM generated topology. We considered two network topologies: each node in the *dense* topology has four links (as in our default setting) on average, whereas the *sparse* setting has two. Figure 15 shows the insertion-only workload. (We further experimented with deleting an additional 20% of the links. Observations were similar and we omit graphs due to space constraints.) The dense network has more derivations than the sparse network: here, *Eager Dense* did not complete after 5 minutes on a 800-link network, whereas *Lazy Dense* finished in under 5 seconds.

Increasing query processing nodes. Next, we increase the number of query processing nodes, while keeping the input dataset constant. Figure 16 shows the results. Per-tuple



(a) Per-tuple Prov. Overhead (B)

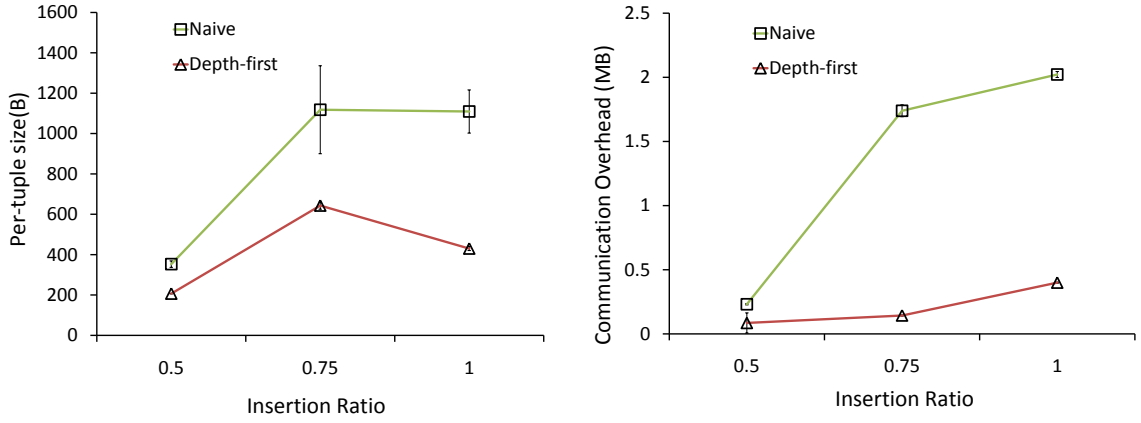
(b) Comm. Overhead (MB)

(c) State within operators (MB)

(d) Convergence time (s)

Figure 16: Varying the number of physical query processing nodes in computing *reachable* query

provenance overhead increases, then eventually levels off, as the number of nodes increases: each node now processes fewer tuples, and the opportunities to absorb or buffer are reduced. More query processors leads to a reduction in query execution latency, per-node communication overhead, and per-node operator state. The increase of latency between 16 and 24 nodes is due to the lower-bandwidth connection between our two subnets. In all cases, *DRed* incurs higher communication overhead and takes longer to complete than our approach.



(a) Per-tuple Prov. Overhead (B)

(b) Comm. Overhead (MB)

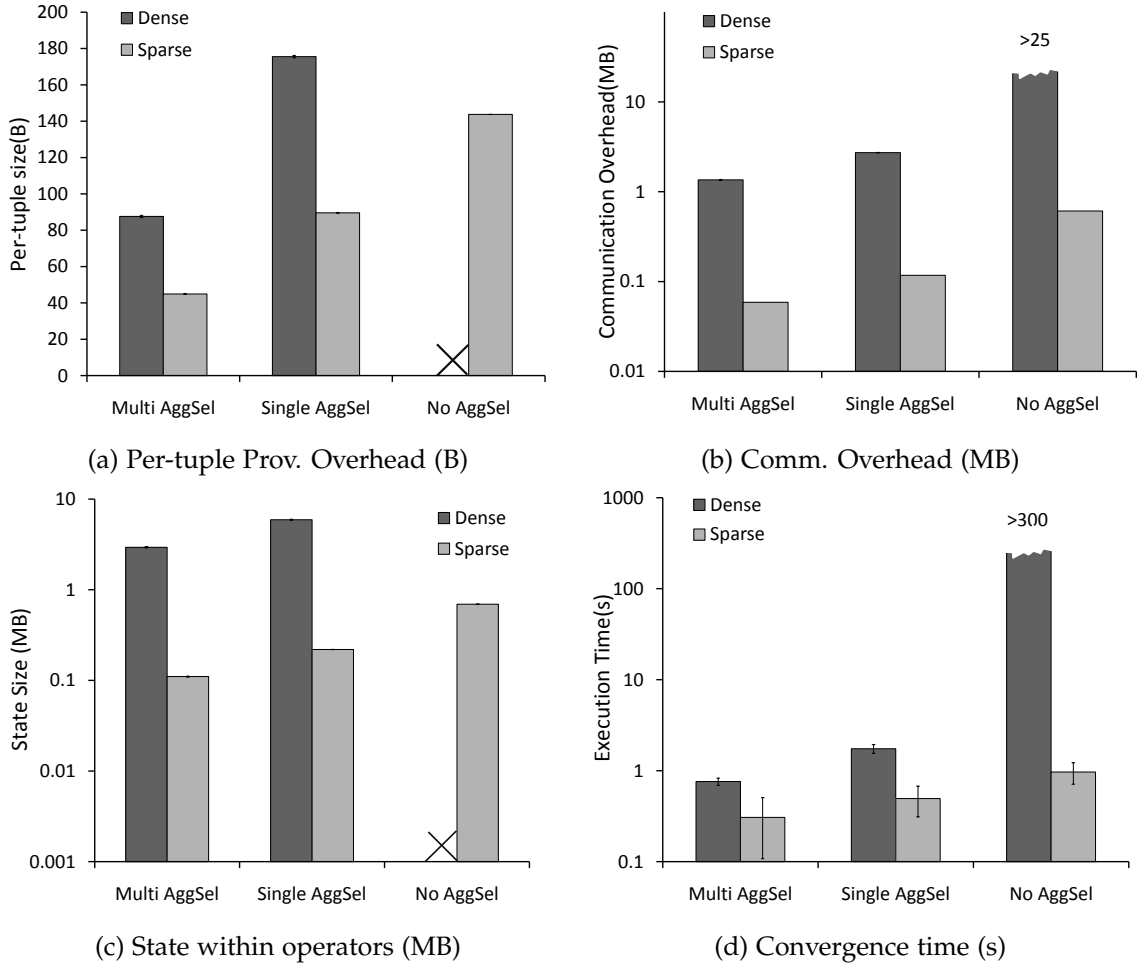
(c) State within operators (MB)

(d) Convergence time (s)

Figure 17: Depth-first search order versus naïve random order on *reachable* query

3.7.4 Provenance BDD Ordering Heuristic

In Figure 17, we compare the performance using our depth-first traversal heuristic, versus naïve merging and ordering of BDDs based on the order of tuple arrival. To better study the performance, we randomize edge arrivals for this experiment. From the figure, the depth-first traversal heuristic saves up to 50% of the provenance overhead. This also results in lower communication overhead and memory footprint. Additionally, execution time remains essentially the same, because the variable reordering process is relatively lightweight and does not affect query processing dataflow. The results show that the execution overhead of applying this heuristics can be offset by the performance gain in memory and communication.



(a) Per-tuple Prov. Overhead (B)

(b) Comm. Overhead (MB)

(c) State within operators (MB)

(d) Convergence time (s)

Figure 18: Aggregate selections performance on *shortestPath* and *cheapestCostPath* query

3.7.5 Multi-aggregate Selection

Figure 18 shows the effectiveness of aggregate selections over the dense and sparse topology of 100 nodes. We experiment with two extensions of the *Shortest Path* query presented in Section 3.1: *Multi AggSel* computes two aggregates (one for shortest path and the other for cheapest cost path); *Single AggSel* minimizes only based on the cheapest cost path. We observe that aggregate selections are most effective in dense topologies, and *Multi AggSel* costs only half as much as *Single AggSel* due to aggressive pruning of the two aggregates simultaneously. Without the use of aggregate selections, all queries are prohibitively expensive, and do not complete within 5 minutes for dense topologies.

3.7.6 Summary of Results

We summarize our experimental results with reference to the contributions of previous sections.

- *Absorption provenance* (Section 3.3) incurs some overhead during insertions and consumes increased memory, versus traditional schemes such as DRed. That increase is offset by improvements in communication overhead and execution times when deletions are part of the workload. Moreover, our concise representation of data provenance is more efficient than an encoding of relative provenance. Most network applications include time-based expiration for state, and hence require frequent deletion processing.

- *Lazy propagation* of derivations (Section 3.4) reduces traffic when there are multiple possible derivations. Lazy propagation results in significant communication cost savings. Given a dense network topology, lazy propagation sped computation by more than an order of magnitude.

- Our heuristic of reordering variables according to a depth-first traversal (Section 3.5) results in up to 50% space and communications savings, with minimal impact on query performance.

- Multiple *aggregate selections* significantly reduce the propagation of tuples during query evaluation (Section 3.6). This is especially true in a dense network with alternative routes, resulting in at least an order of magnitude reduction in communication cost and execution times. While the benefits of aggregate selections have been explored previously in centralized settings, our main contribution here was the extension to a stream model, including support for deletions, and validating that similar benefits are observed in a distributed recursive stream query processor.

3.8 Conclusion

In this chapter, we have proposed novel techniques for distributed recursive stream view maintenance. Our work is driven by emerging applications in declarative networking and

sensor monitoring, where distributed recursive queries are increasingly important. We demonstrated that existing recursive query processing techniques such as DRed [45] are not well-suited for the distributed environment. We then showed how absorption provenance could be used to encode tuple derivability in a compact fashion, then incorporated into provenance-aware operators that are bandwidth efficient and avoid propagating unnecessary information, while maintaining correct answers.

Algorithm 2 Pipelined hash join operator

$HalfPipeIns(u, h_u, p_u, h_j, p_j, \overline{uk})$
Inputs: Update u, build hash table h_u, build provenance table p_u, probe hash table h_j, probe tuple provenance table p_j, join keys $\overline{uk}$.
Output: Update stream u'.

$oldPv := p_u[u.tuple]$
if h_u does not contain $u.tuple$ **then**
  $h_u[u.tuple[\overline{uk}]] := u.tuple$
  $p_u[u.tuple] := u.pv$
else
  $p_u[u.tuple] := p_u[u.tuple] \vee u.pv$
end if
if $oldPv \neq p_u[u.tuple]$ **then**
  for each t in $h_j[u.tuple[\overline{uk}]]$ **do**
    $u'.tuple := u.tuple \bowtie t$
    $u'.type :=$ INS
    $u'.pv := u.pv \wedge p_j[t]$
    Output u'
  end for
end if

$HalfPipeDel(u, h_u, p_u, h_j, p_j, \overline{uk})$
Inputs: Update u, build hash table h_u, build provenance table p_u, probe hash table h_j, probe tuple provenance table p_j, join keys $\overline{uk}$.
Output: Update stream u'.

$oldPv := p_u[u.tuple]$
if h_u contains $u.tuple$ **then**
  $p_u[u.tuple] := p_u[u.tuple] \wedge \neg u.pv$
  if $p_u[u.tuple] = 0$ **then**
    Remove $u.tuple$ from p_u
    Remove $u.tuple[\overline{uk}]$ from h_u
  end if
end if
if $oldPv \neq p_u[u.tuple]$ **then**
  for each t in $h_j[u.tuple[\overline{uk}]]$ **do**
    $u'.tuple := u.tuple \bowtie t$
    $u'.type :=$ DEL
    $u'.pv := u.pv \wedge p_j[t]$;
    Output u'
  end for
end if

$Process(u)$
Inputs: Update u.

if $u.type =$ DEL **then**
  if u is from R^Δ **then**
    $HalfPipeDel(u, h_R, p_R, h_S, p_S, \overline{Rk})$
  else
    $HalfPipeDel(u, h_S, p_S, h_R, p_R, \overline{Sk})$
  end if
else
  if u is from R^Δ **then**
    $HalfPipeIns(u, h_R, p_R, h_S, p_S, \overline{Rk})$
  else
    $HalfPipeIns(u, h_S, p_S, h_R, p_R, \overline{Sk})$
  end if
end if

PipeHashJoin$(R^\Delta, S^\Delta, \overline{Rk}, \overline{Sk}, W_R, W_S)$
Inputs: Update streams R^Δ, S^Δ, join keys $\overline{Rk}, \overline{Sk}$, window evaluation functions W_R, W_S.
Output: Update stream RS^Δ.

Init hash map $h_R : R(\bar{x})[\overline{Rk}] \rightarrow \{R(\bar{x})\}$
Init hash map $h_S : S(\bar{y})[\overline{Sk}] \rightarrow \{S(\bar{y})\}$
Init hash map $p_R : R(\bar{x}) \rightarrow \mathbf{P}(R(\bar{x}))$
Init hash map $p_S : S(\bar{y}) \rightarrow \mathbf{P}(S(\bar{y}))$
while not $EndOfStream(R^\Delta)$ and not $EndOfStream(S^\Delta)$ **do**
  Read an update u from R^Δ or S^Δ
  $Process(u)$
  Let $expired_R$:= the results of calling $W_R(u, p_R)$
  for t in $expired_R$ **do**
    $t.type$:= DEL
    $Process(t)$
  end for
end while

Algorithm 3 Incremental depth-first search algorithm with interval labeling

IncrementalDepthFirst$(e, startInx, endInx, edges)$
Input: Incoming edge e, map from variable to start edge position $startInx$, map from variable to end edge position $endInx$, edge vector $edges$.
Output: Updated $startInx$, $endInx$, and $edges$.

1: $x := e.start; y := e.end$;
2: **if** $startInx[x] < 0$ **then** {no outgoing edge from x}
3: **if** $endInx[x] < 0$ **then** {no incoming edge to x}
4: **if** $startInx[y] < 0$ **then** {no outgoing edge from y}
5: Insert e to the end of $edges$;
6: Update $startInx[x], endInx[x], endInx[y]$;
7: **else** {there exists an outgoing edge from y}
8: **if** the $startInx[y] - 1$ edge of $edges$ ends in y **then** {there exists an incoming edge to y}
9: Insert e to the end of $edges$;
10: Update $startInx[x], endInx[x]$;
11: **else** {no incoming edge to y}
12: Insert e before position $startInx[y]$ of $edges$;
13: Update labels for x, y and all labels with value larger than $startInx[y]$;
14: **end if**
15: **end if**
16: **else** {there exists an incoming edge to x}
17: Insert e after position $endInx[x]$ of $edges$;
18: Update labels for x, y and all labels with value larger than $endInx[x]$
19: **if** $endInx[x] < startInx[y]$ **then** {x's interval appears before y's interval and do not overlap}
20: Move sub-vector $edges[startInx[y]..endInx[y]]$ forward to position $endInx[x] + 1$ of $edges$ and shift other elements;
21: Update all labels according to the new positions in $edges$;
22: **end if**
23: **end if**
24: **else** {there exists an outgoing edge from x}
25: Same procedure as lines 17-22 but do not modify $startInx[x]$;
26: **end if**

Algorithm 4 Aggregate selection sub-module

$AggSel(U^{\Delta}, \overline{uk}, n, agg_1, agg_2, \cdots, agg_n)$
Inputs: Input stream U^{Δ}, grouping keys $\overline{uk}$, number of aggregate functions n, aggregate function $agg_1, agg_2, \cdots, agg_n$.
Output: Stream U'^{Δ}.

1: Init hash map H: $U(\bar{x})[\overline{uk}] \rightarrow \{U(\bar{x})\}$
2: Init hash map P: $U(\bar{x}) \rightarrow$ provenance expressions over $U(\bar{x})$
3: Init hash map B: $U(\bar{x})[\overline{uk}] \rightarrow [1..n] * \{U(\bar{x})\}$
4: **while** not $EndOfStream(U^{\Delta})$ **do**
5: Read an update u from U^{Δ}
6: **if** $u.type$ = INS **then**
7: **if** H does not contain $u.tuple$ **then**
8: $H[u.tuple[\overline{uk}]] := u.tuple$
9: **end if**
10: $P[u.tuple] := u.pv$
11: **if** $oldPv \neq P[u.tuple]$ **then**
12: **for** $i = 1$ to n **do**
13: **if** B does not contain $u.tuple[\overline{uk}]$ **then**
14: $B[u.tuple[\overline{uk}]].i := u.tuple$
15: **else if** $u.tuple$ is better than $B[u.tuple[\overline{uk}]].i$ for agg_i **then**
16: $u'.tuple := B[u.tuple[\overline{uk}]].i$
17: $u'.type := DEL$
18: $u'.pv = P[B[u.tuple[\overline{uk}]].i]$
19: Output u'
20: $B[u.tuple[\overline{uk}]].i := u.tuple$
21: **end if**
22: **end for**
23: **if** $B[u.tuple[\overline{uk}]]$ is updated **then** Output u
24: **end if**
25: **else if** H contains $u.tuple$ **then**
26: $oldPv := P[u.tuple]$
27: Remove $u.pv$ from $P[u.tuple]$
28: **if** $P[u.tuple]$ indicates no derivability **then**
29: Remove $u.tuple$ from P
30: Remove $u.tuple[\overline{uk}]$ from H
31: **end if**
32: **if** $oldPv \neq P[u.tuple]$ **then**
33: **for** $i = 1$ to n **do**
34: **if** $B[u.tuple[\overline{uk}]].i = u.tuple$ **then**
35: Remove $u.tuple$ from $B[u.tuple[\overline{uk}]].i$
36: **for** each tuple t in $H[u.tuple[\overline{uk}]]$ **do**
37: **if** $B[u.tuple[\overline{uk}]].i = null$ or t is better than $B[u.tuple[\overline{uk}]].i$ for agg_i **then**
38: $B[u.tuple[\overline{uk}]].i := t$
39: **end if**
40: **end for**
41: $u'.tuple := B[u.tuple[\overline{uk}]].i$
42: $u'.type = INS$
43: $u'.pv = P[B[u.tuple[\overline{uk}]].i]$
44: Output u'
45: **end if**
46: **end for**
47: **if** $B[u.tuple[\overline{uk}]]$ is updated **then** Output u
48: **end if**
49: **end if**
50: **end while**

Chapter 4

Incremental Re-optimization of Queries: The Declarative Approach

The problem of supporting rapid *adaptation* to runtime conditions during query processing — *adaptive query processing* [35] — is of increasing importance in today's data processing environments. Consider declarative cloud data processing systems [5, 17, 74] and data stream processing [1, 24, 77] platforms, where data properties and the status of cluster compute nodes may be constantly changing. Here it is very difficult to effectively choose a good plan for query execution: data statistics may be unavailable or highly variable; cost parameters may change due to resource contention or machine failures; and in fact a *combination* of query plans might perform better than any single plan. Similarly, in conventional DBMSs there may be a need to perform *self-tuning* so the performance of a query or set of queries can be improved [72].

To this point, query optimization techniques in adaptive query processing systems fall into three general classes: 1) operator-specific techniques that can adapt the order of evaluation for specific combinations of operators rather than the full-fledged space [12, 58]; 2) eddies [10, 33] and related flow heuristics, which are highly adaptive but also continuously devote resources to exploring *all* plans and require fully pipelined execution; 3) approaches that use a cost-based query re-optimizer to re-estimate plan costs and determine whether the system should change plans [54, 56, 72, 85]. Of these, the last is the most flexible, e.g., in that it supports complex query operators like aggregation,

as well as expensive adaptations like data repartitioning across a cluster. Perhaps most importantly, a cost-based engine allows the system to spend the majority of its resources on query execution once the various cost parameters have been properly calibrated. Put another way, it can be applied to highly complex plans and has the potential to provide significant benefit if a cost estimation error was made, but it should incur little overhead if a good plan was chosen. Unfortunately, as today's environments permit more and more data sources to be integrated (as in data integration and environmental monitoring applications), the queries become more and more complex (up to tens of relations not able to complete in seconds), and the stream sources could become bursty, standard cost-based query optimizations could be too expensive to perform frequently (say every 1s where the optimization time exceeds 1s). Unfortunately, to this point cost-based techniques have not been able to live up to their potential, because there lacks a general solution for incremental re-optimization. Here, generality is in the sense that guaranteeing optimality of the plans based on the cost model, just as full-fledged cost optimizers like System-R and Volcano, in handling all types of queries unlimited to specific combinations of operators. Indeed, some previous work has studied the problems before, but they focused on either heuristics or a specific class of queries, rather than exploring the entire search space as in a full-fledged cost optimizer. On the other hand, in order to reduce the overhead of optimizing increasingly complex queries over data streams, we hope to only modify the parts of computations when necessary, i.e., incrementally, across multiple query re-optimizations. By intuition, a small variation of the stream characteristics would only bring small changes to the costs of candidate plans, which may not change the best plan picked; but we still need to maintain the up-to-date status because over time the best plan for data streams could change drastically. It would be great if we always guarantee the best plan returned by a query re-optimizer, during cost-based adaptivity query processing, but incur little overhead when most of the computations could be shared with previous rounds of query re-optimizations.

Our goal in this chapter is to explore whether full-fledged cost-based *incremental* techniques for query re-optimization can be developed, where an optimizer only re-explores query plans whose costs were affected by an updated cardinality or cost value; and whether such incremental techniques could be used to facilitate more efficient adaptivity.

Target Domains. In this chapter we focus on developing incremental re-optimization techniques that we evaluate within a single-node (local) query engine, in two main contexts. 1) We address the problem of adaptive query processing in *data stream management systems* where data may be bursty, and its distributions may vary over time — meaning that different query plans may be preferred over different segments. Here it is vital to optimize frequently based on recent data distribution and cost information, ideally as rapidly as possible. 2) We address query re-optimization in traditional OLAP settings when the same query (or highly similar queries) gets executed frequently, as in a prepared statement. Here we may wish to re-optimize the plan after each iteration, given increasingly accurate information about costs, and we would like this optimization to have minimal overhead.

Approach and Contributions. The main contribution of this chapter is to show how a cost-based full-fledged *incremental* re-optimizer can be developed, and how it can be useful in adaptive query processing scenarios matching the two application domains cited above. Our incremental re-optimizer realizes the basic capabilities of a modern database query optimizer, and could easily be extended to support more advanced features; our main goal is to show that an incremental optimizer following our model can be competitive with a standard optimizer for *initial* query optimization, and significantly faster for *repeated* optimizations. Moreover, in contrast to randomized or heuristics-based [10] optimization methods, we **still guarantee the discovery of the best plan** according to the cost model. Since our work is oriented towards adaptive query processing, we evaluate the system in a variety of settings in conjunction with a basic pipelined query engine for stream and stored data.

We address the problem using a novel approach, which is based on the observation that query optimization is essentially a recursive process involving the derivation and subsequent pruning of state (namely, alternative plans and their costs). If one is to build an *incremental* re-optimizer, this requires preservation of state (i.e., the optimizer memoization table) across optimization runs (as we shall see in the evaluations, this could be under 100M for a reasonably complex query, hence could be loaded in memory in most cases)— but moreover, it must be possible to determine what plans have been *pruned* from

this state, and to re-derive such alternatives and test whether they are still viable.

One way to achieve such "re-pruning" capabilities is to carefully define a semantics for how state needs to be tracked and recomputed in an optimizer. However, we observe that this task of "re-pruning" in response to updated information looks remarkably similar to the database problem of *view maintenance* through aggregation [44] and recursion as studied in the database literature [45]. In fact, recent work [29] has shown that query optimization can itself be captured in recursive datalog. Thus, rather than inventing a custom semantics for incrementally maintaining state within a query optimizer, we instead adopt the approach of developing an incremental re-optimizer expressed *declaratively*.

More precisely, we express the optimizer as a recursive datalog program consisting of a set of rules, and leverage the existing database query processor to actually execute the declarative program. In essence, this is optimizing a query optimizer using a query processor. Our declarative optimizer approaches the performance of conventional procedural optimizers for reasonably-sized queries. It recovers the initial overhead during subsequent re-optimizations by leveraging *incremental view maintenance* [45, 66] ideas. It only recomputes portions of the search space and cost estimates that might be affected by the cost updates. Frequently, this is only a small portion of the overall search space, and hence we often see order-of-magnitude performance benefits.

Our approach achieves pruning levels that rival or best bottom-up (System-R [86]-like) and top-down (Volcano [38, 39]-like) plan enumerations with branch-and-bound pruning. We develop a variety of novel *incremental* and *recursive* optimization techniques to capture the kinds of pruning used in a conventional optimizer, and more importantly, to generalize them to the incremental case. Our techniques are of broader interest to incremental evaluation of recursive queries as well. Empirically, we see updates on only a small portion of the overall search space, and hence we often see order-of-magnitude performance benefits of incremental re-optimization. We also show that our re-optimizer fits nicely into a complete adaptive query processing system, and measure both the performance and quality, the latter demonstrated well in the yielded query plans, of our incremental re-optimization techniques on the Linear Road stream benchmark. We make the following contributions:

- The first cost-based full-fledged query optimizer that prunes yet supports incremental re-optimization.

- A rule-based, declarative approach to query (re)optimization. (As we shall see later, here rule-based means specifying the entire query optimizer declaratively, rather than specifying transformations in rules as in conventional query optimizers [86].). Our approach decouples plan enumerations and cost estimations, relaxing traditional restrictions on search order and pruning.

- Novel strategies to prune the state of an executing recursive query, such as a declarative optimizer: *aggregate selection* with *tuple source suppression*, *reference counting*, and *recursive bounding*.

- A formulation of query re-optimization as an *incremental view maintenance* problem, for which we develop novel incremental algorithms.

- An implementation over a query engine ASPEN developed for recursive stream processing [64, 66], with a comprehensive evaluation of performance against alternative approaches, over a diverse workload.

- Demonstration that incremental re-optimization can be incorporated to good benefit in existing cost-based adaptive query processing techniques [54, 85].

4.1 Declarative Query Optimization

Our goal is to develop infrastructure to adapt an optimizer to support efficient, incrementally maintainable state, and incremental pruning. Our focus is on developing techniques for *incremental state management* for the recursively computed plan costs in the query optimizer, in response to updates to query plan cost information. Incremental update propagation is a very well-studied problem for recursive datalog queries, with a clean semantics and many efficient techniques. Prior work has also demonstrated the feasibility of a declarative query optimizer [29]. Hence, rather than re-inventing procedural solutions we have built our optimizer as a series of recursive rules in datalog, executed

in the query engine that already exists in the DBMS. (We could have further extended to Prolog, but our goal was clean state management rather than a purely declarative implementation. Other alternatives like constraint programming or planning languages do not support incremental maintenance.)

We specify an entire optimizer in three stages and 10 rules (dataflow is illustrated in Figure 20, and full rules are shown in Figure 19). In contrast to work such as [29], our focus is not on formulating every aspect of query optimization in datalog, but rather on formulating those aspects relating to state management and pruning as datalog rules — so we can use incremental view maintenance (delta rules) and sideways information passing techniques, respectively. Other optimizer features that are not reliant on *state* that changes at runtime, such as cardinality estimation, breaking expressions into subexpressions, etc., are specified as built-in auxiliary functions (fn functions). Our abstraction level of the specification helps us to focus on the state management of incremental re-optimization rather than the rewriting of specific functions, which enables advanced features implemented on traditional database systems being reused here as well. In other words, this abstraction level helps us to focus on the *difference* between a re-optimizer and a normal optimizer, and aims to provide an angle through which one can understand how to build a re-optimizer on top of existing procedural ones.

Plan enumeration (SearchSpace**).** Searching the space of possible plans has two aspects. In the *logical phase*, the original query is recursively broken down into a full set of alternative relational algebra subexpressions. (Alternatively, only *left-linear* expressions may be considered [86].) The decomposition is naturally a "top-down" type of recursion: it starts from the original query expression, and then breaks down into subexpressions. The *physical phase* takes as input a query expression from the logical phase, and creates physical plans by enumerating the set of possible physical operators that satisfy any constraints on the output *properties* [39] or "interesting orders" [86] (e.g., the data must be sorted by a particular attribute). Without physical properties, the extension from logical plans to physical plans can be computed either top-down or bottom-up; however, the properties are more efficiently computed in goal-directed (top-down) manner.

Cost estimation (PlanCost**).** This phase determines the cost for each physical plan in the

R1: SearchSpace($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp$) :- Expr($expr, prop$),
Fn_isleaf($expr, false$), Fn_split($expr, prop, \underline{index}, \underline{logOp}, \underline{phyOp}, \underline{lExpr}, \underline{lProp}, \underline{rExpr}, \underline{rProp}$);

R2: SearchSpace($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp$) :-
SearchSpace($-, -, -, -, -, expr, prop, -, -$), Fn_isleaf($expr, false$),
Fn_split($expr, prop, \underline{index}, \underline{logOp}, \underline{phyOp}, lExpr, lProp, rExpr, rProp$);

R3: SearchSpace($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp$) :-
SearchSpace($-, -, -, -, -, -, -, expr, prop$), Fn_isleaf($expr, false$)),
Fn_split($expr, prop, \underline{index}, \underline{logOp}, \underline{phyOp}, \underline{lExpr}, \underline{lProp}, \underline{rExpr}, \underline{rProp}$);

R4: SearchSpace($expr, prop, -,' scan', phyOp, -, -, -, -$)) :-
SearchSpace($-, -, -, -, -, expr, prop, -, -$), Fn_isleaf($expr, true$), Fn_phyOp($prop, \underline{phyOp}$);

R5: SearchSpace($expr, prop, -,' scan', phyOp, -, -, -, -$) :-
SearchSpace($-, -, -, -, -, -, -, expr, prop$), Fn_isleaf($expr, true$), Fn_phyOp($prop, \underline{phyOp}$);

R6: PlanCost($expr, prop, index, logOp, phyOp, -, -, -, -, md, cost$) :-
SearchSpace($expr, prop, index, logOp, phyOp, -, -, -, -$), Fn_scansummary($expr, prop, \underline{md}$),
Fn_scancost($expr, prop, md, \underline{cost}$);

R7: PlanCost($expr, prop, index, logOp, phyOp, lExpr, lProp, -, -, md, cost$) :-
SearchSpace($expr, prop, index, logOp, phyOp, lExpr, lProp, -, -$), Fn_isleaf($lExpr, false$),
PlanCost($lExpr, lProp, -, -, -, -, -, -, -, lMd, lCost$),
Fn_nonscansummary($expr, prop, index, logOp, lMd, -, \underline{md}$),
Fn_nonscancost($expr, prop, index, logOp, phyOp, lExpr, lProp, -, -, md, \underline{localCost}$),
Fn_sum($lCost, null, localCost, \underline{cost}$);

R8: PlanCost($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp, md, cost$) :-
SearchSpace($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp$),
Fn_isleaf($lExpr, false$), $Fn\_isleaf(rExpr, false)$,
PlanCost($lExpr, lProp, -, -, -, -, -, -, -, lMd, lCost$),
PlanCost($rExpr, rProp, -, -, -, -, -, -, -, rMd, rCost$),
Fn_nonscansummary($expr, prop, index, logOp, lMd, rMd, \underline{md}$),
Fn_nonscancost($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp, md, \underline{localCost}$),
Fn_sum($lCost, rCost, localCost, \underline{cost}$);

R9: BestCost($expr, prop, min < cost >$) :-
PlanCost($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp, md, cost$);

R10: BestPlan($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp, md, cost$) :-
BestCost($expr, prop, cost$),
PlanCost($expr, prop, index, logOp, phyOp, lExpr, lProp, rExpr, rProp, md, cost$);

Figure 19: Datalog Rules for the Query Optimizer

search space, by recursively merging the statistics and cost estimates of a plan's subplans. It is naturally a bottom-up type of recursion, as the plan subexpressions must already have been cost-estimated before the plan itself. Here we can encode in a table the mapping from a plan to its cost.

Plan selection (BestPlan**).** As costs are estimated, the program produces the plan that incurs the lowest estimated cost.

In our declarative approach to query optimization, we treat optimizer state as *data* to be queried, use rules to specify what a query optimizer is, and leverage a database query processor to actually perform the computation for the query optimizer. Figure 20 shows a (simplified) execution plan for the datalog rules. As we can see, the declarative program is by nature recursive, and is broken into three stages mentioned above (with Fixpoint operators between stages). Starting from the bottom of the figure, **plan enumeration** recursively generates a *SearchSpace* table containing plan specifications, by decomposing the query and enumerating possible output properties; enumerated plans are then fed into the **plan estimation** component, *PlanCost*, which computes a cost for each plan, by building from leaf to complex expressions; **plan selection** computes a *BestCost* and *BestPlan* entry for each query expression and output property, by aggregating across the *PlanCost* entries.

Example 4. As our driving example, consider a simplified TPC-H Query Q3 with its aggregates and functions removed, called Q3S.

```
SELECT L_orderkey, O_orderdate, O_shippriority
FROM Customer C, Orders O, Lineitem L
WHERE C_mktsegment = 'MACHINERY' and C_custkey = O_custkey and O_orderkey = L_orderkey and
     O_orderdate < '1995-03-15' and L_shipdate > '1995-03-15'
```

□

4.1.1 Plan Enumeration

Plan enumeration takes as input the original query expression as *Expr*, and then generates as output the set of alternative plans. As with many optimizers, it is divided into two levels:

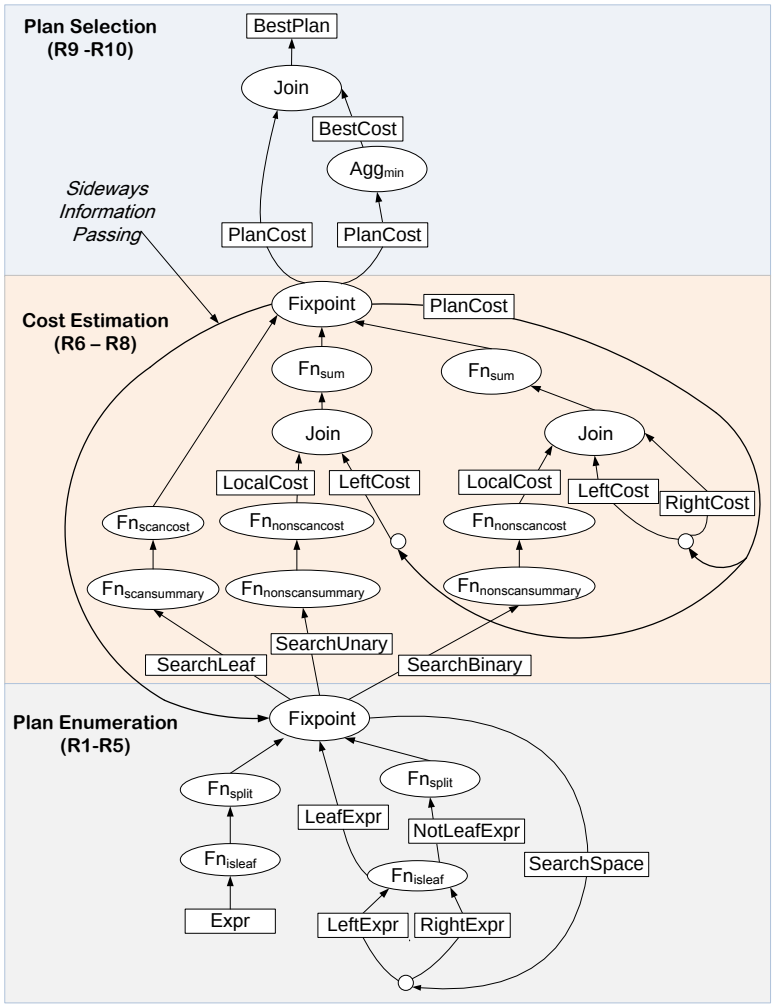


Figure 20: Query plan of our declarative query optimizer. Operators are in ellipses; views are in rectangles. Plan enumeration (SearchSpace) consists of 5 rules, cost estimation (PlanCost) 3 rules, and plan selection (BestPlan) 2 rules. See Figure 19.

Logical search space. The logical plan search space contains all the logical plans that correspond to subexpressions of the original query expression up to any logically equivalent transformations (e.g., commutativity and associativity of join operators). In traditional query optimizers such as Volcano [39], a data structure called an *and-or-graph* is maintained to represent the logical plan search space. Bottom-up dynamic programming optimizers do not need to physically store this graph but it is still conceptually relevant.

Example 5. Figure 21 shows an example and-or-graph for Q3S, which describes a set of alternative subplans and subplan choices using interleaved levels. "AND" nodes represent

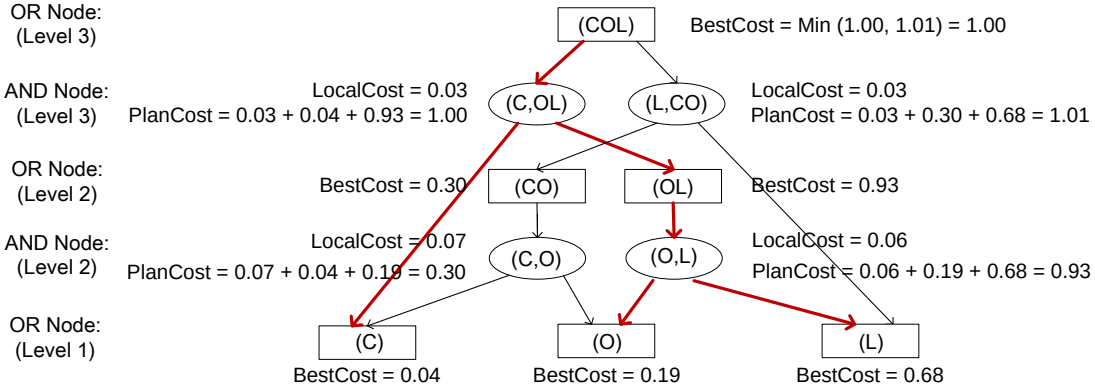


Figure 21: The and-or-graph for Q3S. Red edges denote the best plan. Rectangles and ovals denote "OR" and "AND" nodes respectively. Each "OR" node is labeled with its *BestCost* and each "AND" node is labeled with its *LocalCost* and *PlanCost*.

alternative subplans (typically with join operator roots) and the "OR" nodes represent decision points where the cheapest AND-node was chosen.

□

We capture each of the nodes in a table called *SearchSpace*. In fact, as we discuss next, we supplement this information with further information about the output properties and *physical plan*. (We explain why we combine the results from both stages in Section 4.1.3.)

Physical search space. The physical search space extends the logical one in that it enumerates all the physical operators for each algebraic logical operator. For example, in our figure above, each "AND" node denotes a logical join operator, but it may have multiple physical implementations such as pipelined-hash, indexed nested-loops, or sort-merge join. If the physical implementation is not symmetric, exchanging the left and right child would become a different physical plan. A physical plan not only has a root physical operator, but also a set of physical properties over the data that it maintains or produces; if we desire a set of output properties, this may constrain the physical properties of the plan's inputs.

Example 6. Table 1 shows the *SearchSpace* content for a subset of Figure 21. The AND logical operators are either joins (with 2 child expressions), or tablescans (with selection predicates applied). Each expression *Expr* may have multiple *Index*ed alternatives. *Prop*

*Expr	*Prop	*Index	LogOp	*PhyOp	lExpr	lProp	rExpr	rProp
(COL)	–	1	join	sort-merge	(C)	C_custkey order	(OL)	O_custkey order
(COL)	–	2	join	nested-loop	(L)	L_orderkey index	(CO)	–
(OL)	O_custkey order	1	join	pipe-hash	(O)	–	(L)	–
(CO)	–	1	join	pipe-hash	(C)	–	(O)	–
(CO)	–	1	join	sort-merge	(C)	C_custkey order	(O)	O_custkey order
(O)	O_custkey order	–	scan	local scan	–	–	–	–
(O)	–	–	scan	local scan	–	–	–	–
(L)	L_orderkey index	–	scan	index scan	–	–	–	–
(L)	–	–	scan	local scan	–	–	–	–
(C)	C_custkey order	–	scan	local scan	–	–	–	–
(C)	–	–	scan	local scan	–	–	–	–

Table 1: A simplified *SearchSpace* relation encoding the and-or-graph for Q3S's search space. Primary keys are denoted by *.

and *PhyOp* represent the physical properties of a plan and its root physical operator, respectively.

For instance, expression $SearchSpace(COL)$ encodes an "OR" node with two alternatives, the first "AND" (join) child is $SearchSpace(C, OL)$ and the second is $SearchSpace(L, CO)$. For the first *SearchSpace* tuple, the left expression is C and the right expression is OL. The tuple indicates a Sort-Merge join, whose left and right inputs' physical properties require a sort order based on *C_custkey* and *O_custkey*, respectively. The second alternative uses an Indexed Nested-Loop Join as its physical operator. The left expression refers to the inner join relation indexed on *L_orderkey*, while there are no ordering restrictions on the right expression. □

Our declarative optimizer enumerates both spaces in the same recursive query (see bottom of Figure 20). Given an expression, the $\mathsf{Fn}_{\mathsf{split}}$ function enumerates all the algebraically equivalent rewritings for the given expression, as well as the possible physical operators and their interesting orders (e.g., partitioning, physical location, order, etc). Fixpoint is reached when *Expr* has been decomposed down to leaf-level scan operations over a base relation (checked using the $\mathsf{Fn}_{\mathsf{isleaf}}$ function).

4.1.2 Cost Estimation and Plan Selection

The cost estimation component computes an estimated cost for a physical plan. Given the *SearchSpace* tuples generated in the plan enumeration phase, three datalog rules (R6 - R8) are used to compute *PlanCost* (corresponding to a more detailed version of the "AND" nodes in Figure 21, with physical operators considered and all costs enumerated), and two additional rules (R9 - R10) select the *BestPlan* (corresponding to an "OR" node). Cost estimates are recursively computed by summing up the children costs and operation costs. The computed sum for each physical plan is stored in *PlanCost*.

In addition to the search space, cost estimation requires a set of *summaries* (statistics) on the input relations and indexes, e.g., cardinality of an (indexed) relation, selectivity of operators, data distribution, etc. These summaries are computed using functions $\mathsf{Fn}_{\mathsf{scansummary}}$ and $\mathsf{Fn}_{\mathsf{nonscansummary}}$. The former computes the leaf level summaries over base tables, and the latter computes the output summaries of an operator based on input summaries. Given the statistics, the cost of a plan can be computed by combining factors such as CPU, I/O, bandwidth and energy into a single cost metric. We compute the cost of each physical operator using functions $\mathsf{Fn}_{\mathsf{scancost}}$ and $\mathsf{Fn}_{\mathsf{nonscancost}}$ respectively.

Given the above functions, cost estimation becomes a recursive computation that sums up the cost of children expressions and the root cost, to finally compute a cost for the entire query plan. At each step, $\mathsf{Fn}_{\mathsf{sum}}$ is used to sum up the *PlanCost* of its child plans with *LocalCost*. The particular form of the operation depends on whether the plan root is a leaf node, a unary or a binary operator.

Example 7. To illustrate the process of cost estimation, we revisit Figure 21, which shows a simplified logical search space (omitting physical operators and properties) for our simplified TPC-H Q3S. For every "AND" node, we compute the plan cost by summing up the cost of the join operator, with the best costs of computing its two inputs (e.g., the level 2 "AND" node (C, O) sums up its local cost 0.07, its left best cost 0.04, and its right best cost 0.19, and gets its plan cost 0.30). For every "OR" node, we determine the alternative with minimum cost among its "AND" node children (e.g., the level 3 "OR" node (COL) computes a minimum over its two input plan costs 1.00 and 1.01, and gets its best cost 1.00). After the best cost is computed for the root "OR" node in the graph,

the optimization process is done, and an optimal plan tree is chosen. □

Once the *PlanCost* for every "AND" node are generated, the final two rules compute the *BestCost* for every "OR" node by computing a min aggregate over *PlanCost* of its alternative "AND" node derivations, and output the *BestPlan* for each "AND" node by computing a join between *BestCost* and *PlanCost*.

Note that our approach does not require that complete plans always be generated in order to compute the cost. It allows for sharing sub-plans even across multiple queries as well, because the same logical plans could be *cached* when generating the tuples in *PlanCost*. Indeed, this data-centric query optimization approach can be extended to handle multi-queries, and on top of that, multi-query sharing opportunities could be further explored (e.g., by specifying nonequivalent sharable opportunities into rules as well).

4.1.3 Execution Strategy

Given a query optimizer specified in datalog, the natural question is how to actually execute it. We seek to be general enough to incorporate typical features of existing query optimizers, to rival their performance and flexibility, and to only develop implementation techniques that generalize. We adopt two strategies:

Merging of logical and physical plan enumeration. The logical and physical plan enumeration phases are closely related, and in general one can think of the physical plan as an elaboration of the logical one. As both logical and physical enumeration are top-down types of recursion, and as a logical plan could be regarded as a group representative of various correspondingly physical plans, we can merge the logical and physical enumeration stages into a single recursive query.

As we enumerate each logical subexpression, we simultaneously join with the table representing the possible physical operators that can implement it. This generates the entire set of possible physical query plans. To make it more efficient to generate multiple physical plans from a single logical expression, we use *caching* to memoize the results of $\mathsf{Fn}_{\mathsf{nonscansummary}}$ and $\mathsf{Fn}_{\mathsf{split}}$.

Decoupling of cost estimation and plan enumeration. Cost estimation requires bottom-up evaluation: a cost estimate can *only* be obtained once cost estimates and statistics are

obtained from child expressions. The enumeration stage naturally produces expressions in the order from parent to child, yet estimation must be done from child to parent. We *decouple* the execution order among plan enumeration and cost estimation, making the connections between these two components flexible. For example, some cost estimates may happen before all related plans have been enumerated. Cost estimates may even be used to *prune* portions of the plan enumeration space (and hence also further prune cost estimation itself) in an opportunistic way.

In subsequent sections, we develop techniques to prune and maintain the optimizer state that has no constraints on enumeration order, search order or pruning frequency. Our approach relaxes the traditional restrictions on the search order and pruning techniques in either Volcano-style [39] top-down traversal or System R-style [86] bottom-up dynamic programming approaches. For example, a top-down search may have a depth-first, breadth-first or another order.

We leverage a pipelined push-based query processor to execute the rules in an incremental fashion, which simultaneously explores many expressions. We pipeline the results of plan enumeration to the cost estimation stage without synchronization or blocking.

4.2 Achieving Pruning

In this section, we describe how we can incorporate *pruning* of the search space into pipelined execution of our query optimizer. To achieve this, we use techniques based on the idea of *sideways information passing*, in which the computation of one portion of the query plan may be made more efficient by filtering against information computed elsewhere, but not connected directly by pipelined dataflow. Specifically, we incorporate the technique of aggregate selection [89] from the deductive database literature, which we briefly review; we extend it to perform further pruning; and we develop two new techniques for recursive queries that enable tracking of dependencies and computation of bounds. Beyond query optimization, our techniques are broadly useful in the evaluation of recursive datalog queries. In the next section we develop novel techniques to make these strategies *incrementally maintainable*.

Aggregate selection removes non-viable plans from the optimizer state if they are not

cost-effective, and we show how we can use it to achieve the similar effects to dynamic programming in System R-style optimizers. There we also introduce a novel technique called *tuple source suppression*. Then in the remainder of the section we show how to introduce two familiar notions into datalog execution: namely, *reference counting* that enables us to removes plan subexpressions once all of their parent expressions have been pruned, and *recursive bounding*, which lets the datalog engine incorporate branch-and-bound pruning as in a typical Volcano-style top-down query optimizer. Our solutions are valid for any execution order, take full advantage of the parallel exploration provided by pipelining, and are extensible to parallel or distributed architectures.

4.2.1 Pruning Suboptimal Plan Expressions

Dating back to System-R [86], every modern query optimizer uses dynamic programming techniques (although some via memoization [39]). Dynamic programming is based on the *principle of optimality*, i.e. an optimal plan can be decomposed into sub-plans that must themselves be optimal solutions. This property is vital to optimizer performance, because the same subexpression may appear repeatedly in many parent expressions. Formally:

Proposition 8. *Given a query expression E and property p, consider a plan tree $T\langle E, p\rangle$ that evaluates E with output property p. For this and other propositions, we assume that plans have distinct costs. For cases where plan costs have ties, refer to our extended technical report [60]. Here one such T will have the minimum cost: call that T_{OPT}. Suppose E^s is a subexpression of E, and consider a plan tree $T^s\langle E^s, p^s\rangle$ that evaluates E^s with output property p^s. Again one such T^s will have the minimum cost: call that T^s_{OPT}. T^s is T^s_{OPT} iff T^s is the subtree of T_{OPT}.*

Proof. $\Rightarrow$: Prove by contradiction. If T^s is T^s_{OPT}, then suppose T^s is *not* the subtree of T_{OPT}. Let T'^s be the subtree of T_{OPT}. Since $T^s = T^s_{OPT}$, then we must have $PlanCost(T^s\langle E^s, p^s\rangle) < PlanCost(T'^s\langle E^s, p^s\rangle)$. If we substitute T'^s with T^s in the tree T_{OPT}, we get a new tree T'', with $PlanCost(T''\langle E, p\rangle) < PlanCost(T_{OPT}\langle E, p\rangle)$. Contradiction to the definition of T_{OPT}.

$\Leftarrow$: Prove by contradiction. If T^s is the subtree of T_{OPT}, then suppose T^s is *not* T^s_{OPT}, since plans have distinct costs, we have $PlanCost(T^s\langle E^s, p^s\rangle) > PlanCost(T^s_{OPT}\langle E^s, p^s\rangle)$. If we substitute subtree T^s with subtree T^s_{OPT} in the tree T_{OPT}, we get a new plan tree T',

which has $PlanCost(T'\langle E,p\rangle) < PlanCost(T_{OPT}\langle E,p\rangle)$. Contradiction to the definition of T_{OPT}. □

This proposition ensures that we can safely discard suboptimal *subplans* without affecting the final optimal plan, and the final optimal plan contains exactly the subplans we retain. Consider the and-or-graph of the example query Q3S (Figure 21). The red (bolded) subtree is the optimal plan for the root expression (COL). The subplan of the level 3 "AND" node (L, CO) has suboptimal cost 1.01. If there exists a super-expression containing (COL), then the only viable subplan is the one marked in the figure. State for any alternative subplan for (COL) may be pruned from *SearchSpace* and *PlanCost*. We achieve pruning over both relations as follows.

Pruning *PlanCost* via aggregate selection. Refer back to Figure 20: each *BestCost* tuple encodes the minimum cost for a given query expression-property pair, over all the plans associated with this pair in *PlanCost*. To avoid enumerating unnecessary *PlanCost* tuples, one can wait until the *BestCost* of subplans are obtained before computing a *PlanCost* for a root plan. This is how System R-style dynamic programming works. However, this approach constrains the order of evaluation.

We instead extend a logic programming optimization technique called *aggregate selection* [89], to achieve dynamic programming-like benefits for any arbitrary order of implementation. In aggregate selection, one "pushes down" a selection predicate into the input of an aggregate, such that we can prune results that exceed the current minimum value or are below the current maximum value. In our case (as shown in the middle box of Figure 20), the current best-known cost for any equivalent query expression-property pair is maintained within our Fixpoint operator (which also performs the non-blocked **min** aggregation). We only propagate a newly generated *PlanCost* tuple if its cost is smaller than the current minimum. This does not affect the computation of *BestCost*, which still outputs the minimum cost for each expression-property pair. Since pruning bounds are updated upon every newly generated tuple, there is no restriction on evaluation order. As with pruning strategies used in Volcano-style optimizers, the amount of state pruned varies depending on the order of exploration: the sooner a min-cost plan is encountered, the more effective the pruning is.

Pruning *SearchSpace* via tuple source suppression. Enumeration of the search space will generally happen in parallel with the enumeration of plans. Thus, as we prune tuples from *PlanCost*, we may be able to remove related tuples (e.g., representing subexpressions) from *SearchSpace*, possibly preventing enumeration of their subexpressions and/or costs. We achieve such pruning through *tuple source suppression*, along the arcs indicated in Figure 20. Any *PlanCost* tuples pruned by aggregate selection should also trigger cascading deletions to the *source tuples* by which they were derived from the *SearchSpace* relation. To achieve this, since *PlanCost* contains a superset of the attributes in *SearchSpace*, we simply project out the *cost* field and propagate a deletion to the corresponding *SearchSpace* tuple.

4.2.2 Pruning Unused Plan Subexpressions

The techniques described in the previous section remove *suboptimal plans* for specific expression-property pairs. However, ultimately some *optimal plans* for certain expressions may be unused in the final query execution plan. Consider in Figure 21 the level 2 "AND" node (C, O): this node is not in the final plan because its "OR" node parent expression (CO) does not appear in the final result. In turn, this is because (CO)'s parent "AND" nodes (in this example, just a single plan (L, CO)) does not contribute to the final plan. Intuitively, we may prune an "OR" node if all of its parent "AND" nodes have been pruned, and prune an "AND" node if its direct parent "OR" node has been pruned.

We would like to remove such nodes once they are discovered to not appear in the final optimal plan, which requires a form of *reference counting* within the datalog engine executing the optimizer. To achieve this, every "OR" node, represented by a subexpression and property pair, $\langle E^s, p^s \rangle$, is annotated with a reference *count*.

The reference count of an "OR" node is defined as the number of its **parent** "AND" nodes that are not pruned in the final result (i.e., in *SearchSpace* or *PlanCost*). An "AND" node T^s is pruned either because of suboptimal plans, i.e., $T^s \neq T^s_{OPT}$, or as discussed later in the next section, because it is above a bound, $PlanCost(T^s) > Bound\langle E^s, p^s \rangle$, or as we shall discuss later in this section, because its corresponding expression $\langle E^s, p^s \rangle$ has a reference count of zero. For example, in Table 1, the expression property pair of $\langle (OL), O_{custkey}$ order$\rangle$ has reference count of 1, because it only has one parent plan, which

is the first entry $C \bowtie OL$; on the other hand, the expression property pair of $\langle (C), C_{custkey}$ order $\rangle$ has reference count of 2, because it has two parent plans, which are the first entry, $C \bowtie (OL)$ and the fifth enty, $C \bowtie O$. Below is a proposition about reference counting:

Proposition 9. *Given a query expression E with output property p: let T^s be a plan tree for E's subexpression E^s with property p^s. E^s with property p^s has reference count of zero, i.e., $RefCount\langle E^s, p^s\rangle = 0$ iff T^s is* not *a subtree of the optimal plan tree for the query E with property p, $T_{OPT}\langle E, p\rangle$.*

Proof. $\Rightarrow$: Prove by contradiction. If $RefCount\langle E^s, p^s\rangle = 0$, then any plan tree whose root is the AND node representing the parent of $\langle E^s, p^s\rangle$ is pruned. Suppose T^s is a subtree of $T_{OPT}\langle E, p\rangle$, then consider the immediate parent node of T^s in T_{OPT}, let it be $T^{s'}$. According to Proposition 8, $T^{s'} = T^{s'}_{OPT}$. This means it has the minimal cost to subexpression $\langle E^{s'}, p^{s'}\rangle$, is on the optimal tree of the final optimal plan, but has been pruned. Contradiction.

$\Leftarrow$: Prove by contradiction. If T^s is *not* a subtree of T_{OPT}, according to Proposition 8, T^s is not T^s_{OPT}. Hence, $PlanCost(T^s_{OPT}) < PlanCost(T^s)$. Suppose T^s has the reference count other than zero, then it must have at least a parent plan that has not been pruned. Suppose that parent plan is $T'^{s'}$. If we substitute T^s with T^s_{OPT} in $T'^{s'}$, we get a plan tree $T''^{s'}$ that has a smaller cost than $T'^{s'}$. Hence $T'^{s'}$ can be pruned in the final result because it is suboptimal to $T''^{s'}$. Contradiction. □

The proposition ensures that a plan with a reference count of zero can be safely deleted, and a plan with a positive reference count must be the subtree of the optimal plan tree. Note that a deleted plan may make more reference counts to drop to zero, hence the deletion process may be recursive. Our reference counting scheme is more efficient than the *counting* algorithm of [45], which uses a count representing the *total number of derivations* of each tuple in bag semantics. Our count represents the number of *unique parent plans from which a subplan may be derived*, and can typically be incrementally updated in a single recursive step (whereas **counting** often requires multiple recursive steps to compute the whole derivation count).

Our reference counting mechanism complements the pruning techniques of aggregate selection. Following an insertion (exploration) or deletion (pruning) of a *SearchSpace* tu-

r1: ParentBound$(lExpr, lProp, bound - rCost - localCost)$:-
Bound$(expr, prop, bound)$, BestCost$(rExpr, rProp, rCost)$,
LocalCost$(expr, prop, index, lExpr, lProp, rExpr, rProp, -, localCost)$;
r2: ParentBound$(rExpr, rProp, bound - lCost - localCost)$:-
Bound$(expr, prop, bound)$, BestCost$(lExpr, lProp, lCost)$,
LocalCost$(expr, prop, index, lExpr, lProp, rExpr, rProp, -, localCost)$;
r3: MaxBound$(expr, prop, max < bound >)$:- ParentBound$(expr, prop, bound)$;
r4: Bound$(expr, prop, min < minCost, maxBound >)$:- BestCost$(expr, prop, minCost)$,
MaxBound$(expr, prop, maxBound)$;

Figure 22: Datalog rules to express bounds computation

ple, we update the reference counts of relevant tuples accordingly; cascading insertions or deletions of *SearchSpace* (and further *PlanCost*) tuples may be triggered because their reference counts may be raised above zero (or dropped to zero). Finally, the optimal plan computed by the query optimizer is unchanged, but more tuples in *SearchSpace* and *PlanCost* are pruned. Indeed, by the end of the process, the combination of aggregate selection and reference counts ensure *SearchSpace* and *PlanCost only* contain those plans that are on the final optimal plan tree. Such "garbage collection" greatly reduces the optimizer's state and the number of data items that must be updated incrementally, as described in Section 4.3.

4.2.3 Full Branch-and-Bound Pruning

Our third innovation is to implement the full effect of *branch-and-bound pruning*, as in top-down optimizers like Volcano, during cost estimation of physical plans. Branch-and-bound pruning uses *prior exploration* of related plans to prune the exploration of new plans: a physical plan for a subexpression is pruned if its cost already exceeds the cost of a plan for the equivalent subexpression, *or* the cost of a plan to any parent, grandparent, or other ancestor expression of this subexpression. Unfortunately, branch-and-bound pruning assumes a single-recursive descent execution thread in its enumeration. Our ultimate goal is to find a branch-and-bounding solution independent of the search order, and able to support parallel enumeration (e.g., a different thread to the plan enumeration or cost estimation process).

Previous work [29] has shown that it is possible to do a limited form of branch-and-

bound pruning in a declarative optimizer, by initializing a bound based on the cost of the parent expression, and then pruning subplan exploration whenever the cost has exceeded an *equivalent* expression. This can actually be achieved by our aggregate selection approach.

We seek to generalize this to prune against the *best* known bound for an expression-property pair — which may be from a plan for an equivalent expression, or from any ancestor plan that *contains* the subplan corresponding to this expression-property pair. (Recall that there may be several parent plans for a subplan: this introduces some complexity as each parent plan may have different cost bounds, and at certain point in time we may not know the costs for some of the parent plans.) The bound should be continuously *updated* as different parts of the search space are explored via pipelined execution. In this section, we assume that bounds are initialized to infinity and *monotonically decreasing*. In Section 4.3.3 we will relax this requirement.

Our solution, *recursive bounding*, creates and updates a single recursive relation *Bound*, whose values form the *best-known* bound on each expression-property pair (each "OR" node). This bound is the minimum of 1) known costs of any equivalent plans; 2) the highest bound of any parent plan's expression-property pair, which in turn is defined recursively in terms of parents of this parent plan. Figure 22 shows how we can express the bounds table using recursive datalog rules. *ParentBound* propagates cost bounds from a parent expression-property pair to child expression-property pairs, through *LocalCost*, while the child bound also takes into account the cost of the local operator, and the best cost from the sibling side. *MaxBound* finds the highest of bounds from parent plans, and *Bound* maintains the minimum bounding information derived from *BestCost* or *MaxBound*, allowing for more strict pruning.

Given the definition of *Bound*, we can reason about the viability of certain physical plans below:

Proposition 10. *Given a query expression E with desired output property p: let T^s be a plan subtree that produces E's subexpression E^s and yields property p^s. T^s has a cumulative cost PlanCost that is larger than the Bound of subexpression E^s and yields property p^s, i.e., $PlanCost(T^s) > Bound\langle E, p\rangle$ iff T^s is* not *a subtree of the optimal plan tree for the expression E and property p,*

$T_{OPT}\langle E, p\rangle$.

Proof. $\Rightarrow$: Prove by contradiction. Suppose $PlanCost(T^s) > Bound\langle E, p\rangle$. And suppose T^s is a subtree of the optimal plan tree T_{OPT}, according to Proposition 8, T^s is T^s_{OPT}. Hence, $PlanCost(T^s_{OPT}) > Bound\langle E, p\rangle$. Now we prove by induction that if T^s is a subtree of T' of subexpression $E^{s'}$ and property $p^{s'}$, then $PlanCost(T'_{OPT}) > Bound\langle E^{s'}, p^{s'}\rangle$.

First, the base case $PlanCost(T^s_{OPT}) > Bound\langle E^s, p^s\rangle$ holds. Now, if $PlanCost(T^s_{OPT}) > Bound\langle E^s, p^s\rangle$, according to the definition of *Bound* where $Bound\langle E^s, p^s\rangle = \min(BestCost(T^s), maxBound\langle E^s, p^s\rangle)$, we have $Bound\langle E^s, p^s\rangle = MaxBound\langle E^s, p^s\rangle < PlanCost(T^s_{OPT})$. According to the definition of *MaxBound*, $MaxBound\langle E^s, p^s\rangle = \max(ParentBound\langle E^s, p^s\rangle)$, therefore, for any arbitrary sibling OR node of T^s, $Sib(T^s)$, and corresponding parent AND node of T^s, $Par(T^s)$ of expression $E^{s'}$ and property $P^{s'}$, we have $Bound\langle E^{s'}, p^{s'}\rangle$ - $BestCost(Sib(T^s))$ - $LocalCost(Par(T^s)) \leq MaxBound\langle E^s, p^s\rangle < PlanCost(T^s_{OPT})$. Hence, $Bound\langle E^{s'}, p^{s'}\rangle < BestCost(Sib(T^s)) + BestCost(T^s) + LocalCost(Par(T^s))$. Because this holds for any sibling and parent, we have $Bound\langle E^{s'}, p^{s'}\rangle < \min(BestCost(Sib(T^s)) + BestCost(T^s) + LocalCost(Par(T^s))) = BestCost(Par(T^s)) = PlanCost(Par(T^s)_{OPT})$.

By applying this induction we prove that if T^s is a subtree of T' of subexpression $E^{s'}$ and property $p^{s'}$, then $PlanCost(T'_{OPT}) > Bound\langle E^{s'}, p^{s'}\rangle$. Since T^s is a subtree of the optimal plan tree T_{OPT}, hence $PlanCost(T_{OPT}) > Bound\langle E, p\rangle$. However, as T_{OPT} is the root, it has no parent, according to the definition of *Bound*, it should have $Bound(T_{OPT}) = PlanCost(T_{OPT})$. Contradiction.

$\Leftarrow$: Prove by contradiction. Suppose T^s is *not* a subtree of T_{OPT}, according to Proposition 8, T^s is not T^s_{OPT}, hence, $PlanCost(T^s) > PlanCost(T^s_{OPT})$. Suppose $PlanCost(T^s) \leq Bound\langle E^s, p^s\rangle$, then we have $PlanCost(T^s_{OPT}) < PlanCost(T^s) \leq Bound\langle E^s, p^s\rangle$. According to the definition of *Bound*, $Bound\langle E^s, p^s\rangle = \min(BestCost(T^s), maxBound\langle E^s, p^s\rangle)$, we have $PlanCost(T^s_{OPT}) < BestCost(T^s)$, contradiction to the definition of T^s_{OPT}.

□

Based on Proposition 10, recursive bounding may safely remove any plan that exceeds the bound for its expression-property pair, and after the process, the plans retained are exactly the ones on the optimal plan tree. Indeed, with our definition of the bounds, this strategy is a generalization of the aggregate selection strategy. However, bounds are

recursively defined here and a single plan cost update may result in a number of changes to bounds for others.

Overall the execution flow of pruning *PlanCost* and *SearchSpace* via recursive bounding is similar to that described in the aggregate selection strategy. Specifically, *PlanCost* is pruned inside the Fixpoint operator, where an additional comparison check $PlanCost < Bound$ is performed before propagating a newly generated *PlanCost*. Updates over other *Bound* tuples derived from a given *PlanCost* tuple are computed separately. *SearchSpace* is again pruned via sideways information passing where the pruned *PlanCost* tuples are directly mapped to deletions over *SearchSpace*.

4.3 Incremental Re-Optimization

The previous section described how we achieve pruning at a level comparable to a conventional query optimizer, without being constrained to the standard data and control flow of a top-down or bottom-up procedural implementation. In this section, we discuss *incremental* maintenance during both query optimization and re-optimization. In particular, we seek to incrementally update not only the state of the optimizer, but also the state that affects pruning decisions, e.g., reference counts and bounds.

Initial query optimization takes a query expression and data summaries, and produces a set of tables encoding the plan search space and cost estimates. During execution, pruning bounds will always be monotonically decreasing. Now consider *incremental* re-optimization, where the optimizer is given updated cost (or cardinality) estimates based on the information collected at runtime after partial execution. This scenario commonly occurs in adaptive query processing, where we monitor execution and periodically re-optimize based on the updated status. For simplicity, our discussion of the approaches assumes that a single *cost parameter* (operator estimated cost, output cardinality) changes. Our approach is able to handle multiple such changes simultaneously.

Given a change to a cost parameter, our goal is in principle to re-evaluate the costs for all affected query plans. Some of these plans might have previously been pruned from the search space, meaning they will need to be re-enumerated. Some of the pruning bounds might need to be adjusted (possibly even raised), as some plans become more

expensive and others become cheaper. As the bounds are changed, we may in turn need to re-introduce further plans that had been previously pruned, or to remove plans that had previously been viable. This is where our declarative query optimizer formulation is extremely helpful: we use *incremental view maintenance* techniques to only recompute the necessary results, while guaranteeing correctness.

Incremental maintenance enabled via datalog. From the declarative point of view, initial query optimization and query re-optimization can be considered roughly the same task, if the data model of the datalog program is **extended** to include updates (insertions, deletions and replacements). Indeed, incremental query re-optimization can be specified using a delta rules formulation like [45]. This requires several extensions to the database query processor to support **direct processing of deltas**: instead of processing standard tuples, each operator in the query processor must be extended to process delta tuples encoding changes. A delta tuple of a relation R may be an insertion ($R[+x]$), deletion ($R[-x]$), or update ($R[x \rightarrow x']$). For example, a new plan generated in *SearchSpace* is an insertion; a pruned plan in *PlanCost* is a deletion; an updated cost of *BestCost* is an update.

We extend the query processor following standard conventions from continuous query systems [59] and stream management systems [77]. The extended query operators consume and emit deltas largely as if they were standard tuples. For stateful operators, we maintain for each encountered tuple value a (possibly temporarily negative) *count*, representing the cumulative total of how many times the tuple has been inserted and deleted. Insertions increment the count and deletions decrement it; counts may temporarily become negative if a deletion is processed out of order with its corresponding insertion, though ultimately the counts converge to nonnegative values, since every deletion is linked to an insertion. We would prune a plan when it goes to zero, and re-insert it when it is not zero any more. As in a stream system, one has no idea whether the count will change in the future, hence the count scheme ensures consistency in the end. Finally, a tuple only affects the output of a stateful operator if its count is positive.

Upon receiving a series of delta tuples, every query operator 1) updates its corresponding state, if necessary; 2) performs internal computations such as predicate evaluation over the tuple or against the state; 3) constructs a set of output delta tuples. Joins

follow the rules established in [45]. For aggregation operators that compute minimum (or maximum) values, we must further extend the internal state management to keep track of *all* values encountered — such that, e.g., we can recover the "second-from-minimum" value. If the minimum is deleted, the operator should propagate an update delta, replacing its previous output with the next-best-minimum for the associated group (and conversely for maximum).

Challenge: recomputation of pruned state. While datalog allows us to propagate updates through rules, a major challenge is that the pruning strategies of Section 4.2 are achieved *indirectly*. In this section we detail how we incrementally re-create pruned state as necessary. We first show how we incrementally maintain the output of aggregate selection and "undo" tuple source suppression. Then we describe how to incrementally adjust the reference counts and maintain the pruned plans. Finally, we show how we can incrementally modify the pruning bounds and the affected plans.

4.3.1 Incremental Aggregate Selection

Aggregate selection [89], as briefly reviewed in Section 4.2.1, prunes state against bounds and does not consider how incremental maintenance might change the bound itself. In order to tackle the incremental case, one can extend the non-incremental aggregate selection approach. Recall that in exploiting aggregate selection to our non-incremental problem, we push down a selection predicate, *PlanCost* $<$ *BestCost*, within the Fixpoint operator that generates *PlanCost*. To illustrate how this works, consider how we may revise *BestCost* and *BestPlan* after encountering an insertion, deletion or update to *PlanCost*. There are four possible cases:

1. Upon an *insertion PlanCost[+c]*, set *BestCost* to **min** (c, current *BestCost*).

2. Upon a *deletion PlanCost[-c]*, set *BestCost* to the next-best *PlanCost* iff the current *BestCost* is equal to c.

3. Upon a *cost update PlanCost[c→c']*, if $c < c'$, set *BestCost* to **min** (c', next-best *PlanCost*) iff the current *BestCost* is equal to c.

4. Upon a *cost update PlanCost[c→c']*, if $c > c'$, if the current *BestCost* is equal to c, then set *BestCost* to c'; else set *BestCost* to **min** (c', current *BestCost*).

Recall that each *PlanCost* tuple denotes a newly computed cost associated with a physical plan, and a *BestCost* tuple denotes the best cost that has been computed so far for this physical plan's expression-property pair. We update *BestCost* based on the current state of *PlanCost*. In Cases 1 and 4, we can directly compute updates to *BestCost*. In Cases 2 and 3, we rely on the fact that the aggregate operator preserves all the computed, even pruned *PlanCost* tuples (as described previously), so it can find the "next best" value even if the minimum is removed. In our implementation we use a priority queue to store the sorted tuples.

We may also need to re-introduce tuples in *SearchSpace* that were suppressed when they led to *PlanCost* tuples that were pruned, we achieve this by propagating an insertion (rather than deletion as in Section 4.2.1) to the previous stage.

4.3.2 Incremental Reference Checking

Once we have updated the set of viable plans for given expressions in the search space, we must consider how this impacts the viability of their subplans: we must incrementally update the reference counts on the child expressions to determine if they should be left the same, re-introduced, or pruned. As before, we simplify this process and make it order-independent through the use of incremental maintenance techniques.

We incrementally and recursively maintain the reference counts for each expression-property pair whenever an associated plan in the *PlanCost* relation is inserted, deleted or updated. When a new entry is inserted into *PlanCost*, we increment the count of each of its child expression-property pairs; similarly, whenever an existing entry is deleted from *PlanCost*, we decrement each child reference count. Replacement values for *PlanCost* entries do not change the reference counts, but may recursively affect the *PlanCost* entries for super-expressions. Whenever a count goes from 0 to 1 (or drops from 1 to 0) we recompute (prune, respectively) all of the physical plans associated with this expression-property pair.

If we combine this strategy with aggregate selection, only the best-cost plan needs to be pruned or re-introduced (all others are pruned via aggregate selection). The aggregate operators internally maintain a record of all *PlanCost* tuples they have received as input, so "next-best" plans can be retrieved if the best-cost entry gets deleted or updated to a higher cost value. During incremental updates, we only propagate changes affecting the old and new best-cost plan and all recursively dependent plans.

4.3.3 Incremental Branch-and-bounding

Perhaps the most complex pruning technique to adapt to incremental maintenance is the branch-and-bound pruning structure of Section 4.2.3: as new costs for any operation are discovered, we must recursively recompute the bounds for all super-expressions. As necessary we then update *PlanCost* and *SearchSpace* tuples based on the updated bounds. Recall from Figure 22 that the *Bound* relation's contents are computed recursively based on the **max** bounds derived from parent plans; and also based on the **min** values for equivalent plan costs. Hence, an update to *LocalCost* or *BestCost* may affect the entries in *Bound*. Here we again rely in part on the fact that *Bound* is a recursive query and we can incrementally maintain it, then use its new content to adjust the pruning. We illustrate the handling of cost updates by looking at what happens when a cost *increases*.

Suppose a plan's *LocalCost* increases. As a consequence of the rules in Figure 22, the *ParentBound* of this plan's children may increase due to rules r1 and r2. *MaxBound* is then updated by r3 to be the maximum of the *ParentBound* entries: hence it may also increase. As in the previous cases, the internal aggregate operator for *ParentBound* maintains all input values; thus, it can recompute the new minimum bound and output a corresponding update from old to new value. Finally, as a result of the updated *ParentBound*, *Bound* in r4 may also increase. The process may continue recursively to this plan's descendant expression-property pairs, until *Bound* has converged to the correct bounds for all expression-property pairs. In practice, the increase of a suboptimal's *LocalCost* could be cached and only be propagated when it really becomes relevant.

Alternatively, suppose an expression-property pair's *BestCost* estimate increases (e.g., due to discovering the machine is heavily loaded). This may trigger an update to the

corresponding entry in *Bound* (via rule r4). Moreover, via rules r1 and r2, an update to this bound may affect the bounds on the parent expression, i.e., *ParentBound*, and thus affecting any expression whose costs were pruned via *ParentBound*.

The cases for handling cost *decreases* are similar (and generally simpler). Sometimes, in fact, we get simultaneous changes in different directions. Consider, for instance, that an expression's cost bound may increase, as in the previous paragraph. At the same time, perhaps the expression-property pair's *ParentBound* may decrease. Any equivalent plan (sibling expression) for our original expression-property pair is bounded *both* by the bounds of sibling expressions and parents. As *ParentBound* decreases, *MaxBound* and *Bound* may also potentially decrease through r3 and r4. The results are guaranteed to converge to the best of the sibling and parent bounds.

So far we focused only on how to update bounds given updated cost information; of course, there is the added issue of updating the pruning results. Recall in Section 4.2.3 that we evaluate the following predicate ϕ before propagating a newly generated *PlanCost* value: if $PlanCost < Bound$ then set *Bound* to *PlanCost*. When *PlanCost* or *Bound* is updated, we can end up in any of 3 cases:

1. Upon an update to a plan cost entry, i.e., *PlanCost[+c]*, *PlanCost[-c]* or *PlanCost[c→c']*: if predicate ϕ's result changes from false to true, then emit an insertion of the *PlanCost* tuple; otherwise if ϕ's result changes from true to false, then emit a deletion. Incrementally update the corresponding *Bound* entry, including its aggregated cost value, as a result.

2. Upon an update on *Bound[b→b']* where $b < b'$: for those tuples t in *PlanCost* where $b < t.cost < b'$, re-insert t into *PlanCost* and re-insert t's counterpart in *SearchSpace* to *undo* tuple source suppression.

3. Upon an update on *Bound[b→b']* where $b > b'$: for those tuples t in *PlanCost* where $b > t.cost > b'$, prune tuple t from *PlanCost* and delete t's counterpart from *SearchSpace* via tuple source suppression.

Indeed, the first step is similar to incremental aggregate selection. The main difference is that here the condition check is not on *BestCost* but rather on *Bound*. Essentially we want

to incrementally update *Bound* based on the current bounding status, hence a sorted list of *PlanCost* tuples needs to be maintained.

An interesting observation of Cases 2 and 3 is that an update on *Bound* may affect the pruned or propagated plans as well. If a bound is raised, it may re-introduce previously pruned plans; if a bound is lowered, it may incrementally prune previously viable plans. If incremental aggregate selection is used, then only the optimal plan among the pruned plans needs to be revisited. *SearchSpace* is again updated via sideways information passing.

4.4 Experimental Results

In this section, we discuss the implementation and evaluate the performance of our declarative optimizer: both versus other strategies, and as a primitive for adaptive query processing. (Note that we reuse existing adaptive techniques from [54, 85]; our focus is on showing that incremental re-optimization *improves* these.)

We implemented the optimizer as 10 datalog rules (see Figure 19) plus 8 external functions (involving histograms, cost estimation, and expression decomposition). Our goal was to implement as a proof of concept the **common core** of optimizer techniques — not an exhaustive set. We executed the optimizer on top of the Aspen system's query engine [66]. To support the pruning and incremental update propagation features in this chapter, we added approximately 20K lines of code to the query engine. In addition, we developed a plan generator to translate the declarative optimizer into a dataflow graph as in Figure 20. Our experiments were performed on a single local node.

For comparison, we implemented in Java a Volcano-style top-down query optimizer and a System-R-style dynamic programming optimizer, which reuse the histogram, cost estimation, and other procedural user-defined-functions as our declarative optimizer. This enables apple-to-apple comparison because all the differences between our declarative optimizer and these procedural counterparts are in dataflow, enumeration order, execution strategy rather than performance differences on specific computations like cost estimations, or histogram generations, etc. Indeed, all the advanced features developed on top of the traditional query optimizers could be applied to our optimizer with no spe-

cial treatment, hence we focus on the difference of dataflow of execution and incremental capabilities here.

We also built a variant of our declarative optimizer that only uses the pruning strategies of the Evita Raced declarative optimizer [29]. Wherever possible we used common code across the implementations to ensure fair comparison.

Experimental Workload. For **repeated optimization** scenarios we use TPC-H queries, with data from the TPC-H and skewed TPC-D data generators [78] (Scale Factor 1). Here, the scale factor is completely unimportant here as we measure the optimization time rather than execution time, when the data is big it just takes more time to do cost estimations, whose costs are common for all approaches, and hence uninteresting for comparisons), with Zipfian skew factor 0 for the latter. Note that here we evaluate single-round optimization, hence even a tiny improvement (e.g., from 1.2s to 0.8s) could save cumulative overhead in adaptive query processing for data streams (e.g., re-optimize every 1s).

We focused on the single-block SQL queries: Q1, Q3, Q5, Q6 and Q10. (Q1 and Q6 are aggregation-only queries; Q3 joins 3 relations; Q10 joins 4; and Q5 joins 6 relations). Our experiments showed that Q1, Q3, and Q6 are all simple enough to optimize that they are simple enough that they completed in under 80msec on all implementations. (The declarative approach tended to add 10-50msec to these settings, as it has higher initialization costs.)

Thus we focus our presentation on reasonably complex queries, e.g., with more than 3-way joins. To create greater query diversity, we modified the 6-way and larger join queries by removing aggregation — we constructed a simplified query Q5S. Finally, to test scale-up to larger queries, we manually constructed an eight-way join query, Q8Join, as the largest possible query to join relations in TPC-H, Q8JoinS. For **adaptive stream processing** we used the Linear Road benchmark [9]: We modified its biggest query, SegToll, by unfolding nested relations, into its simplified version SegTollS. We show our queries Q5, Q5S, Q8Join and SegTollS in Table 2.

Experimental Methodology. We aim to answer four questions:

- Can a declarative query optimizer perform at a rate competitive with procedural opti-

Q5: SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'AMERICA' and o_orderdate $\geq$ '1993-01-01' and o_orderdate $<$ '1994-01-01' GROUPBY n_name;
Q5S: SELECT n_name, l_extendedprice * (1 - l_discount) FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'AMERICA' " and o_orderdate $\geq$ '1993-01-01' and o_orderdate $<$ '1994-01-01';
Q8Join: SELECT c_name, p_name, ps_availqty, s_name, o_custkey, r_name, n_name, sum(l_extendedprice * (1 -l_discount)) FROM orders, lineitem, customer, part, partsupp, supplier, nation, region WHERE o_orderkey = l_orderkey and c_custkey = o_custkey and p_partkey = l_partkey and ps_partkey = p_partkey and s_suppkey = ps_suppkey and r_regionkey = n_regionkey and s_nationkey = n_nationkey GROUPBY c_name, p_name, ps_availqty, s_name, o_custkey, r_name, n_name;
SegTollS: SELECT r1_expway, r1_dir, r1_seg, COUNT(distinct r5_xpos) FROM CarLocStr [size 300 time] as r1, CarLocStr [size 1 tuple partition by expway, dir, seg] as r2, CarLocStr [size 1 tuple partition by caid] as r3, CarLocStr [size 30 time] as r4, CarLocStr [size 4 tuple partition by carid] as r5 WHERE r2_expway = r3_expway and r2_dir = 0 and r3_dir = 0 and r2_seg $<$ r3_seg and r2_seg $>$ r3_seg - 10 and r3_carid = r4_carid and r3_carid = r5_carid and r1_expway = r2_expway and r1_dir = r2_dir and r1_seg = r2_seg GROUPBY r5_carid, r2_expway, r2_dir, r2_seg;

Table 2: Queries modified based on TPC-H and LinearRoad benchmark queries used in our experiments

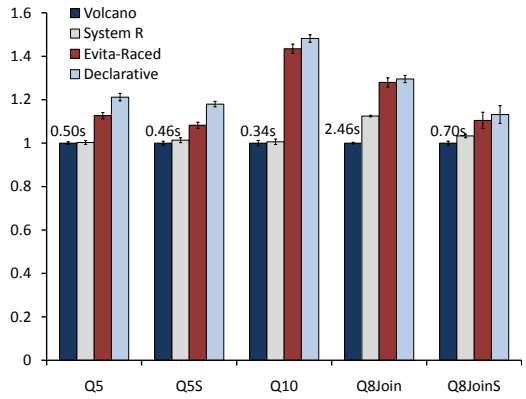
mizers, for 4-way-join queries and larger?

- Does incremental query re-optimization show running time and search space benefits versus non-incremental re-optimization, for repeated query execution-over-static-data scenarios?

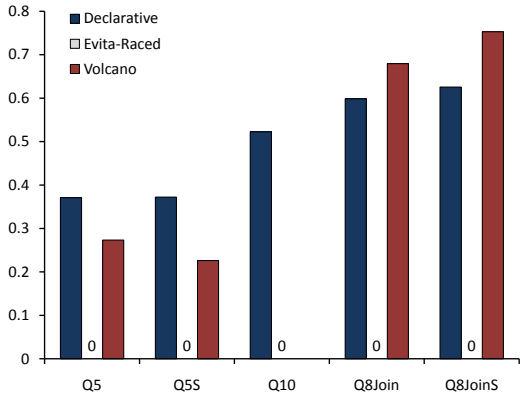
- How does each of our three pruning strategies (aggregate selection, reference counting, and recursive bounding) contribute to the performance?

- Does incremental re-optimization improve the *overall* performance of cost-based adaptive query processing techniques for streaming?

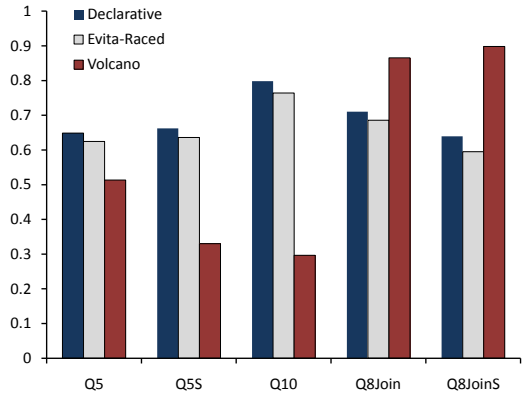
The TPC-H benchmark experiments are conducted on a single local desktop machine: a dual-core Intel Core 2 2.40GHz with 2GB memory running 32-bit Windows XP Profes-



(a) Optimization time (normalized to Volcano)



(b) Pruning ratio: plan table entries



(c) Pruning ratio: plan alternatives

Figure 23: Performance comparison for initial query optimization, across different optimizer architectures

sional, and Java JDK 1.6. The Linear Road benchmark experiments are conducted on a single server machine: a dual-core Intel Xeon 2.83GHz with 8 GB memory running 64-bit Windows Server Standard. Performance results are averaged across 10 runs, and 95% confidence intervals are shown. We mark as 0 any results that are exactly zero.

4.4.1 Declarative Optimization Performance

Our initial experiments focus on the question of **whether our declarative query optimizer can be competitive with procedural optimizers** based on the System R-style (bottom-up enumeration through dynamic programming) and Volcano-style (top-down enumeration with memoization and branch-and-bound pruning) models. To show the value of the pruning techniques developed in this chapter, we also measure the performance when

our engine is limited to the pruning techniques developed in Evita Raced [29] (where pruning is only done against logically equivalent plans for the same output properties). Recall that all of our implementations share the same procedural logic (e.g., histogram derivation, cost estimation); their differences are in search strategy, dataflow, and pruning techniques.

We begin with a running time comparison among Volcano-style, System R-style, and declarative implementations (one using our sideways information passing strategies, and one based on the Evita Raced pruning heuristics) — shown in Figure 23 (a). This graph is normalized against the running times for our Volcano-style implementation (which is also included as a bar for visual comparison of the running times). Actual Volcano-style running times are shown directly above the bar. Observe from the graph that the Volcano-style strategy is always the fastest, though System R-style enumeration often approaches its performance due to simpler (thus, slightly faster) exploration logic. Our declarative implementation is not quite as fast as the dedicated procedural optimizers, with an overhead of 10-50%, but this is natural given the extra overhead of using a general-purpose engine and supporting incremental update processing. The Evita Raced-style declarative implementation is marginally faster in this setting, as it does less pruning. We shall see in later experiments that there are significant benefits to our more aggressive strategies during re-optimization — which is our focus in this work.

To better understand the performance of the different options, we next study their effectiveness in *pruning* the search space. We divide this into two parts: 1) pruning of expression-property entries in the plan table, such that we do not need to compute and maintain *any* plans for a particular expression yielding a particular property; 2) pruning of *plan alternatives* for a particular expression-property pair. In terms of the and-or graph formulation of Figure 21, the first case prunes or-nodes and the second prunes and-nodes. We show these two cases in Figure 23 parts (b) and (c). We omit the System R-style optimizer from this discussion, as it uses a dynamic programming-based pruning model that is difficult to directly compare.

Part (b) shows that our declarative implementation achieves pruning of approximately 35-80% of the plan table entries, resulting in large reductions in state (and, in many cases, reduced computation). We compare with the strategies used by Evita Raced, which we

can see never prunes plan table entries, and with our Volcano-style implementation. Observe that our pruning strategies — which are quite flexible with respect to order of processing — are often more effective than the Volcano-style strategy, which is limited to top-down enumeration with branch-and-bound pruning. (All pruning strategies' effectiveness depends on the specific order in which nodes are explored: better pruning is achieved when inexpensive options are considered early. However, in the common case, high levels of pruning are observed.)

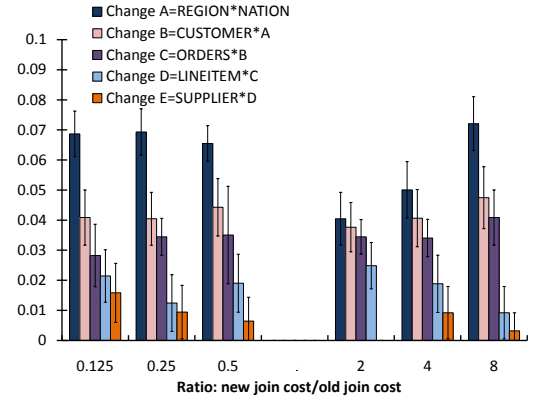
Part (c) looks in more detail at the number of alternative query plans that are pruned: here our declarative implementation prunes approximately 55-75% of the space of plans. It exceeds the pruning ratios obtained by the Evita Raced strategies by around 4-8%, and often results in significantly greater pruning than Volcano-style.

Our conclusions from these experiments are that, even for initial query optimization "from scratch," a declarative optimizer can be performance-competitive with a procedural one — both in terms of running time and pruning performance. Moreover, given that our plan enumeration and pruning strategies are completely decoupled, we plan to further study whether there are effective heuristics for exploring the search space in our model.

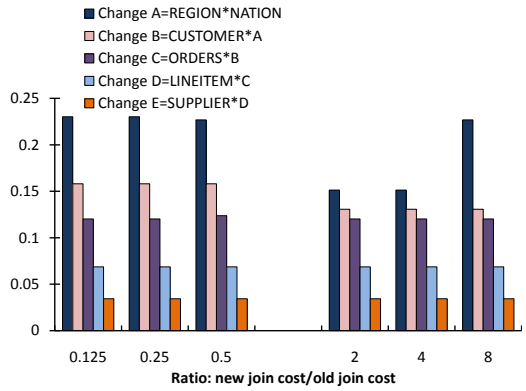
4.4.2 Incremental Re-optimization

Now that we understand the relative performance of our declarative optimizer in a conventional setting, we move on to study how it handles incremental changes to costs. A typical setting in a non-streaming context would be to improve performance during the repeated execution of a query, such as for a prepared statement where only a binding changes. The question we ask is how expensive — given a typical update — it is to re-optimize the query and produce the new, predicted-optimal plan.

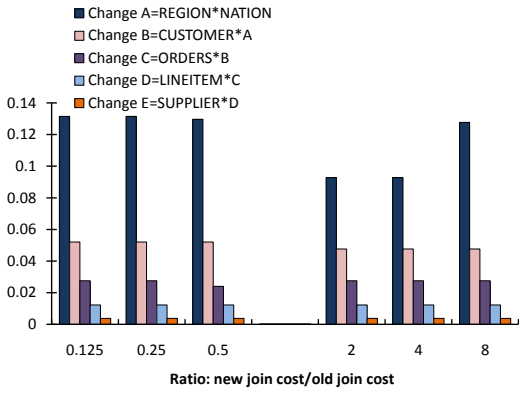
Note that there exists no comparable techniques for incremental cost-based re-optimization, so we compare the gains versus those of re-running a complete optimization (as is done in [54, 85]). In these experiments, we consider running time — versus the running times for the best-performing initial optimization strategy, namely that of our Volcano-style implementation — as well as how much of the total search space gets re-enumerated. We consider re-optimization under "microbenchmark"-style simulated changes to costs, for



(a) Optimization time (normalized to Volcano)



(b) Update ratio: plan table entries



(c) Update ratio: plan alternatives

Figure 24: Performance during incremental re-optimization of TPC-H Q5 — change to join selectivity estimate

synthetic updates as well as observed execution conditions over skewed data. We measured performance across the **full suite of queries** in our workload. However, due to space constraints, and since the results are representative, we focus our presentation on query Q5.

4.4.2.1 Synthetic Changes to Subplan Costs

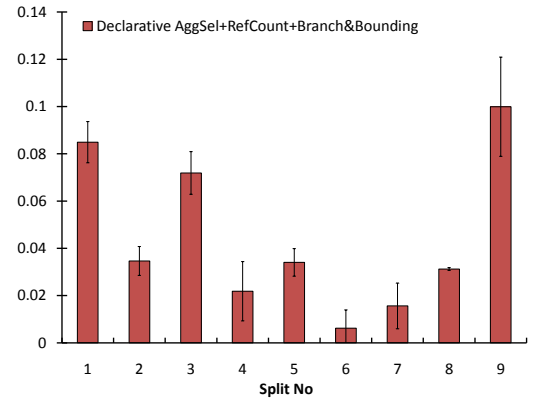
We first simulate what happens if we discover that an operator's output is not in accordance with our original selectivity estimates. Figures 24 (a)-(c) show the impact of synthetically injecting changes for each join expression's selectivity, and therefore the *PlanCost* of the related plans and their super-plans. For conciseness in the graph captions, we assign a symbol with each expression, e.g., the first join $Region \bowtie Nation$ is expression

A, and the second join expression combines the output of A with data from the *Customer* table, yielding $B = Customer \bowtie A$. We expect that changes to smaller subplans will take longer to re-optimize, and changes to larger subplans will take less time (due to the number of recursive propagation steps involved). We separately plot the results of changing each expression's selectivity value, as we change it along a range from 1/8 the predicted size through 8 times larger than the predicted size. Running times in part (a) are plotted relative to the Volcano-style implementation's performance: we see that the speedups are at least a factor of 12, when the lowest-level join cost is updated; going up to over 300, when the topmost join operator's selectivity is changed. In general the speedups confirm that larger expressions are cheaper to update. We can observe from these last two figures that we recompute only a small portion of the search space.

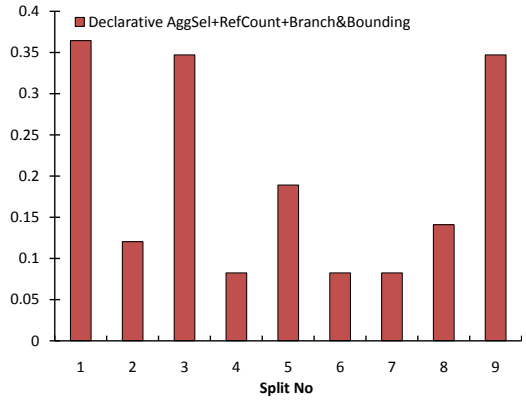
4.4.2.2 Changes based on Real Execution

We now look at what happens when costs are updated according to an actual query execution. We took TPC-H Q5 and to gain better generality, we divided its input into 10 partitions (each having uniform distribution and independent variables) that would result in equal-sized output. We optimized the query over one such partition, using histograms from the TPC-H dataset. Then we ran the resulting query over different partitions of **skewed data** (Zipf skew factor 0.5, from the Microsoft Research skewed TPC-D generator [78]); each of which exhibits different properties. At the end we re-optimized given the cumulatively observed statistics from the partition. We performed re-optimization of each of such interval, given the current plan and the revised statistics.

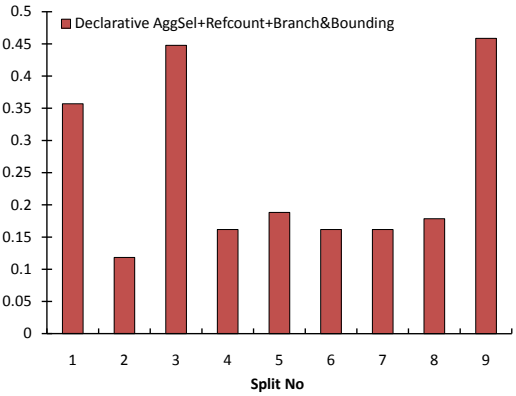
Figure 25 (a) shows the execution times for each round of incremental re-optimization, normalized against the running time of Volcano-style. We see that, as with the join re-estimation experiments of Figure 24, there are speedups of a factor of 10 or greater. In terms of throughput, the Volcano-style model takes 500msec to perform one optimization, meaning it can perform 2 re-optimizations per second; whereas our declarative incremental re-optimizer can achieve 20-60 optimizations per second, and it can respond to changing conditions in 10-100msec. Again, Figure 25 (b) and (c) show that the speedup is due to significant reductions in the amount of state that must be recomputed.



(a) Optimization time (normalized to Volcano)



(b) Update ratio: plan table entries

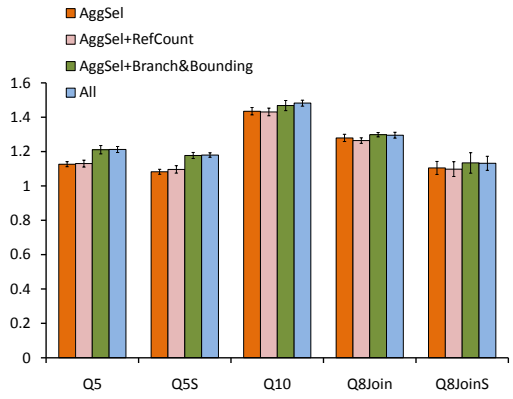


(c) Update ratio: plan alternatives

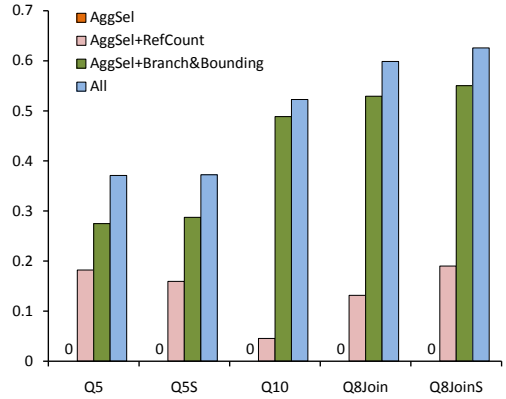
Figure 25: Performance during incremental re-optimization of TPC-H Q5 — updates to costs based on real execution over skewed data

4.4.3 Contributions of Pruning Strategies

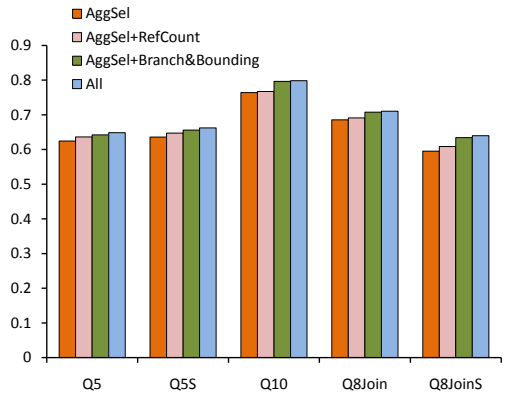
Here we investigate how each of our pruning and incremental strategies from Sections 4.2 and 4.3 contribute to the overall performance of our declarative optimizer. We systematically considered all techniques individually and in combination, unless they did not make sense (e.g., reference counting must be combined with one of the other techniques, and branch-and-bound requires aggregate selection to perform pruning of the search space). See Figure 26, where *AggSel* refers to aggregate selection with source tuple suppression; *RefCount* refers to reference counting; and *Branch-and-Bounding* refers to recursive bounding. We consider aggregate selection in isolation and in combination with the other techniques individually and together. (We also considered the case where none of the pruning techniques are enabled: here running times were over 2 minutes, due to a complete lack



(a) Optimization time (normalized to Volcano)



(b) Pruning ratio: plan table entries



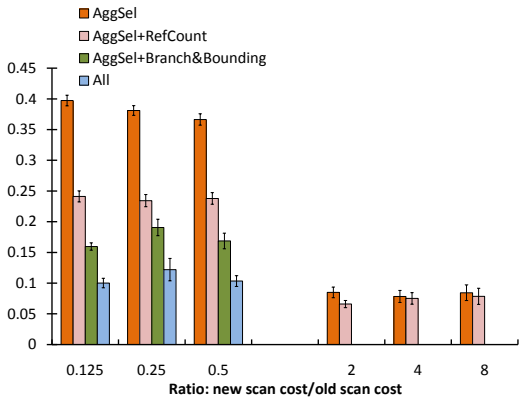
(c) Pruning ratio: plan alternatives

Figure 26: Performance breakdown of pruning techniques for initial optimization, across full query workload

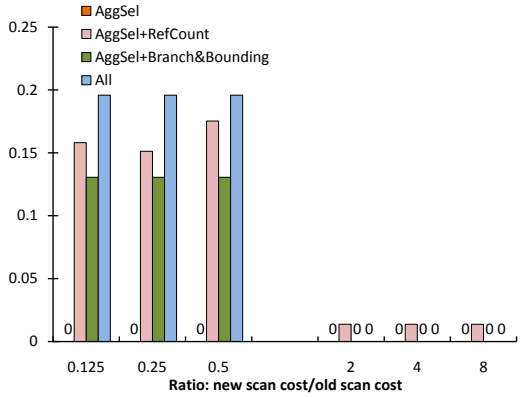
of pruning. We omit this from the graphs.)

Figures 26 (a)-(c) compare the three pruning strategies when performing initial optimization on various TPC-H queries. It can be observed that each of the pruning techniques adds a small bit of runtime overhead (never more than 10%) in this setting, as each requires greater computation and data propagation. Parts (b) and (c) show that each technique adds greater pruning capability, however.

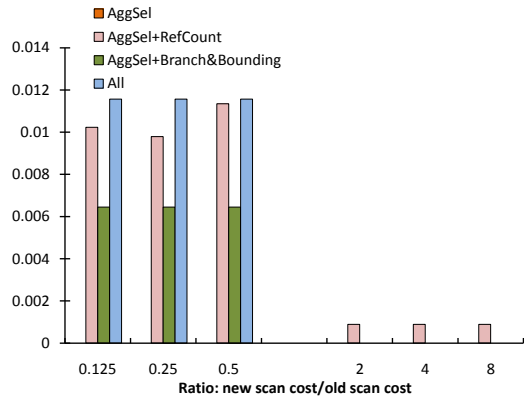
Once we move to the incremental setting — shown in Figures 27 (a)-(c) for query Q5 and changes to the *orders* table, over different cost estimate changes — we see significant benefits in running time as well as pruned search space. Note that in contrast to our other graphs for incremental re-optimization, plots (b) and (c) isolate the amount of pruning performed, rather than showing the total state updated. We see here that our different



(a) Optimization time (normalized to Volcano)



(b) Pruning ratio: plan table entries



(c) Pruning ratio: plan alternatives

Figure 27: Performance breakdown of pruning techniques during incremental re-optimization of Q5 when *Orders* has updated scan cost

techniques work best in combination, and that each increases the amount of pruning.

Bookkeeping overhead. Our pruning strategies enable incremental updates to be supported with relatively minor space overhead: even for the largest query (Q8Join), the total optimizer state was under 100MB. This includes all the book-keeping overhead introduced in our approaches.

4.4.4 Incremental Re-optimization for Adaptive Stream Processing

A major motivation for our work was to facilitate better *cost-based adaptive query processing*, especially for continuous optimization of *stream queries*. Here we show the benefits of incremental re-optimization on a streaming LinarRoad Benchmark workload, by measuring both the optimization times and the execution times during multi-round adaptive

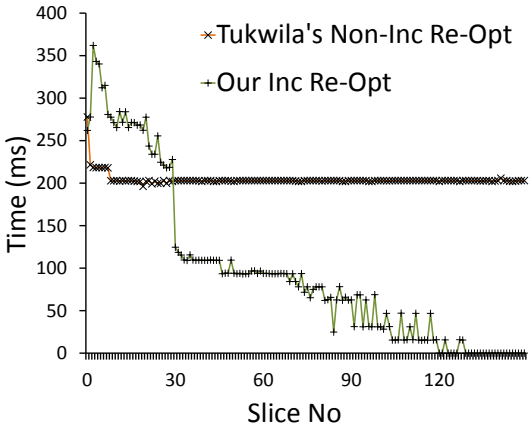


Figure 28: AQP Query Re-optimization Time of SegTollS Query

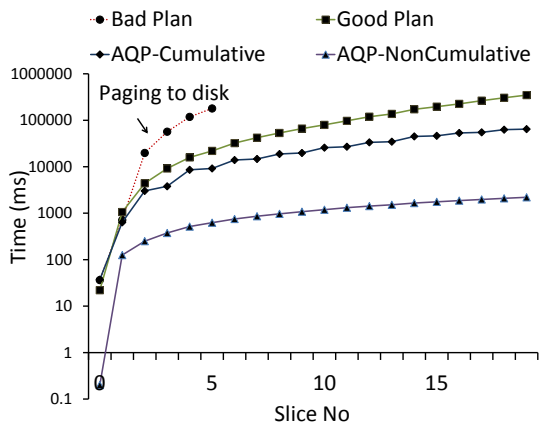


Figure 29: AQP Query Execution Time of SegTollS Query

query processing. This final set of experiments shows how our techniques can be used within a full cost-based adaptive query processing framework, one based on the model of [54] where the optimizer periodically pauses plan execution, forming a "split" point from which it may choose a new plan and continue execution. In general, if we change plans at a split point, there is a challenge of determining how to combine state across the splits. In contrast to [54] we chose *not* to defer the cross-split join execution until the end; rather, we used CAPE's *state migration* strategy [85] to perform the delta computations (e.g., joining tuples across different splits) by migrating the state from the prior plan into the current one, recomputing intermediate state, and computing delta tuples at the new plan. Note that this data-partitioned model could be combined with other cost-based adaptive schemes such as [56, 72] as well.

We use the standard LinearRoad Benchmark [9] data generator to synthesize data streams whose characteristics frequently change over time, resulting in opportunities for plan adaptations. A stream of car location reports, called CarLocStr, is generated by the simulator and we use the default setting of two expressways (L=2). In [7], it maintains a list of continuous queries specified in Continuous Query Language (CQL) with the same window semantics as in our system, and we adapt one of its query, SegToll, which computes the toll for each segment *not* in accidents. We unfold the nested views in SegToll, and generate a simplified query, SegTollS, displayed in Table 2, which contains 5-way joins with multiple windowed aggregates.

Per Slice	Re-Opt Time	N.C. Exec Time	Total Time
1s	5.75s	2.20s	7.95s
5s	1.23s	6.82s	8.05s
10s	0.63s	13.35s	13.98s

Table 3: Frequency of Adaptation (20 sec stream)

We show the performance of our re-optimizer over the SegTollS query over streaming LinearRoad data workloads. The query re-optimization time and query execution time are shown in Figure 28 and Figure 29 respectively.

In Figure 28, we measure the performance of query re-optimization when adapting every 1 second over the input stream. We focus on the optimization times of *non-cumulative* data computation within each slice to emphasize the impact of the plan picked by an optimizer on the performance of new data computation. We measure our incremental re-optimization overhead versus a non-incremental Tukwila-style optimizer [54]. Our re-optimization time starts off from around 0.4s and drops towards 0 till 150 slices, whereas a non-incremental Tukwila-style optimizer holds onto 0.2s of re-optimization time throughout the stream processing. Indeed, we just showed 150 slices, which are the worst scenarios because the initial stream characteristics fluctuate wildly before they have been stabilized inside fixed-size windows). When we take a longer horizon (e.g., 300 slices), we have observed that the re-optimization time remains nearly zero (hence we did not show them on the figure). This is because the stream rarely adapts a plan after it stabilizes (only when stream statistics are off wildly) and the stream usually *converges* to a single plan where our re-optimization is near zero (a non-incremental approach would be too expensive to compute frequently).

We also measure the execution times of the adapted plans picked by our re-optimizer in Figure 29. Here our goal is to have the optimizer start with zero statistical information on the stream data, and find a sequence of plans whose running time equals or betters the *single best static plan* that it would pick given complete information of the stream statistics (e.g., histograms). We also obtain a "bad" plan computed by a standard optimizer assuming it only has partial knowledge about the stream, e.g., the first several seconds of the stream information. Figure 29 shows that our incremental AQP scheme provides **superior performance to the single best plan** ("good single plan"), if we re-optimize every

1 second. This is because the adaptive scheme has a chance to "fit" its plan to the local characteristics of whatever data is currently in the window, rather than having to choose one plan for all data.

A natural question then is where the "sweet spot" is between query execution overhead and query optimizer overhead, such that we can achieve best performance. We measured, for a variety of slice sizes, the total running time of query re-optimization and execution time *over each new slice of data* (not considering the additional overhead of state migration, which depends on how similar the plans are). We can see the results in Table 3: there are significant gains in shrinking the interval from 10sec to 5sec, but little more gain to be had in going down to 1sec. Figure 29 helps explain this: as we execute and re-optimize, the overhead of a non-incremental re-optimizer remains constant (about 200 msec each time), whereas the incremental re-optimization time drops off rapidly, going to nearly zero. This means that the system has essentially *converged on a plan* and that new executions do not affect the final plan.

4.4.5 Experimental Conclusions

We summarize our evaluation results by providing the answers to the questions raised earlier in this section. First, our declarative query optimizer performs respectably when compared to a procedural query optimizer, for initial optimization: despite the overhead of starting up a full query processor, it gets within 10-50% of the running times of a dedicated optimizer. It more than recovers this overhead during incremental re-optimization, where — across a variety of queries and changes — it typically shows an order-of-magnitude speedup, or better. These gains are largely due to having to re-enumerate a much smaller space of plans than a full re-optimization. In addition, we find that our pruning techniques developed in Section 4.2 and Section 4.3 each contribute in a meaningful way to the overall performance of incremental re-optimization. Finally, we show that our incremental re-optimization techniques help cost-based adaptation techniques provide *finer-grained* adaptivity and hence better overall performance. Overhead decreases as the system *converges on a single plan*.

4.5 Conclusion

To build large-scale, pipelined query processors that are reactive to conditions across a cluster, we must develop new adaptive query processing techniques. This chapter represents the first step towards that goal: namely, a fully cost-based architecture for incrementally re-optimizing queries. We have made the following contributions:

- A rule-based, declarative approach to query (re)optimization in adaptive query processing systems.
- Novel optimization techniques to prune the optimizer state: *aggregate selection*, *reference counting*, and *recursive bounding*.
- A formulation of query re-optimization as an incremental view maintenance problem, for which we develop novel incremental algorithms to deal with insertions, deletions and updates over runtime cost parameters.
- An implementation over the ASPEN query engine [66], with a comprehensive evaluation of performance against alternative approaches, over a diverse workload, showing order-of-magnitude speedups for incremental re-optimization.

Chapter 5

Incremental Re-optimization of Queries: The Procedural Approach

In the previous chapter, we studied how to address incremental re-optimization under declarative frameworks. In this chapter, we aim to leverage the insights we gained from the declarative perspective solving the problem, and apply them to traditional procedural cost-based query optimization engines. Examples of such engines include bottom-up style query optimizers [47, 86] and top-down style query optimizers [39]. These traditional cost-based procedural query optimizers were originally designed for single-pass query optimization, and we strive to understand how to adapt them to be able to incrementally re-optimize. The motivation of this study comes from several observations. First, traditional procedural query optimization frameworks are widely used in the industry and academia and are already extremely complex, hence, it is usually easier to incorporate the new features into the existing codebase than rewriting everything from scratch. Second, by restricting ourselves to traditional procedural frameworks, we can understand the minimal changes needed to make an existing query optimizer incremental.

We aim at understanding incremental query re-optimization approaches for full-fledged cost-based query optimization frameworks, hence our work is different from other solutions proposed in the past, such as operator-specific techniques (e.g., [12]), eddies-based flow-heuristic techniques (e.g., [10]) or heuristics-based re-optimization techniques (e.g.,

[56]). We make the following contributions in this work:

- We define the cost-based procedural incremental query re-optimization problem (in Section 5.1).
- We leverage the lessons learned from the declarative perspective and present incremental re-optimization algorithms for both bottom-up and top-down style architectures, and discuss optimizations to improve recomputation and book-keeping costs (in Section 5.2).
- We analyze the worst-case bounds for re-computations of AND and OR nodes during incremental query re-optimization (in Section 5.3).
- We empirically evaluate our procedural incremental re-optimization algorithms and study the performance differences between top-down and bottom-up style incremental re-optimization, and the difference to their non-incremental counterparts. We study the effects of different optimizations, and understand the empirical results of the ratio of recomputed AND and OR nodes during incremental re-optimization (in Section 5.4).

5.1 Problem Statement of Cost-based Incremental Query Re-optimization

In this section, we illustrate the notion of query optimization, query re-optimization and incremental query re-optimization, all in a full-fledged cost-based fashion. We first review the task of query optimization.

Definition 11. The task of a full-fledged cost-based query optimizer is to take a query expression E as input, and output the optimal physical plan tree T that can implement the execution of this query expression with the least estimated cost. In order to generate this optimal physical plan tree T, a cost-based query optimizer needs to compute an and-or-graph G, where each OR node represents alternative ways of splitting a subexpression (subexpression-physical property pair if we consider a physical and-or-graph), and each AND node represents one way of splitting this subexpression, using an operator that

concatenates one or more smaller subexpressions, which can in turn be represented by OR nodes. Hence, AND nodes and OR nodes are interleaved in the and-or-graph. This graph must have a single OR node representing the original expression E, and several OR nodes as leaf terminals representing the smallest subexpressions that the original expression can be split into.

The process of computing the optimal plan tree T for the original expression E is the process of enumerating and computing the *estimated costs* on the physical and-or-graph. Specifically, each AND node is associated with an estimated cost indicating the operational cost of performing the computation over its subexpressions (e.g., the cost of performing a join over two subexpressions), we call this cost $LocalCost\langle E^s, p^s, i\rangle$ if this AND node represents the i-th way of splitting the expression E^s with the output physical property p^s (physical property needs to be used when considering physical plans). Once each enumerated AND node has computed its *LocalCost*, it could compute the cumulative cost of its subtrees with its own *LocalCost*. We call this cumulative cost *PlanCost*. For every expression E^s and physical property p^s and i-th way of splitting, we have $PlanCost\langle E, p, i\rangle = LocalCost\langle E, p, i\rangle + BestCost\langle E^l, p^l\rangle + BestCost\langle E^r, p^r\rangle$, where we have $BestCost\langle E, p\rangle = \mathbf{min}_i PlanCost\langle E, p, i\rangle$ associated with each OR node. The output of a query optimizer is thus the plan tree T with the minimum estimated cost, $min_p BestCost\langle E, p\rangle$ where E is the original query expression.

Now we define query re-optimization. The goal of a query re-optimizer is to compute the optimal plan tree T' with the minimum estimated cost, $min_p BestCost'\langle E, p\rangle$ of the original query expression E, where $BestCost'$ is the new plan cost of the root level OR node: $BestCost'\langle E, p\rangle = BestCost\langle E, p\rangle + \Delta BestCost\langle E, p\rangle$. The inputs to a query re-optimization algorithm is a series of changes to the and-or-graph (caused by cardinality changes of subexpressions, selectivity changes of intermediate operators, etc). In theory, these changes can be modeled as a set of delta changes to the local plan costs of intermediate AND nodes, $\{\Delta LocalCost\langle E, p, i\rangle\}$, and/or a set of delta changes to the scan costs of leaf OR nodes, $\{\Delta ScanCost\langle E, p\rangle\}$. Hence, we can regard the set of $\Delta LocalCost$ and $\Delta ScanCost$ are inputs and the plan tree T' with the minimum estimated cost, $\mathbf{min}_p BestCost'\langle E, p\rangle$, as the output of the re-optimizer.

Suppose we also define $PlanCost'$ as the new plan costs of AND nodes: $PlanCost'\langle E, p, i\rangle = PlanCost\langle E, p, i\rangle + \Delta PlanCost\langle E, p, i\rangle$. Now the delta plan cost of an AND node of the triple (expression E, property p, index i) is:

$$\begin{aligned}
\Delta PlanCost\langle E, p, i\rangle &= \Delta LocalCost\langle E, p, i\rangle + \Delta BestCost\langle E^l, p^l\rangle + \Delta BestCost\langle E^r, p^r\rangle \\
&= \Delta LocalCost\langle E, p, i\rangle + \mathbf{min}_j PlanCost'\langle E^l, p^l, i\rangle - \mathbf{min}_j PlanCost\langle E^l, p^l, j\rangle \\
&+ \mathbf{min}_k PlanCost'\langle E^r, p^r, k\rangle - \mathbf{min}_k PlanCost\langle E^r, p^r, k\rangle \\
&= \Delta LocalCost\langle E, p, i\rangle + \mathbf{min}_j(PlanCost\langle E^l, p^l, i\rangle + \Delta PlanCost\langle E^l, p^l, i\rangle) \\
&- \mathbf{min}_j PlanCost\langle E^l, p^l, j\rangle + \mathbf{min}_k(PlanCost\langle E^r, p^r, k\rangle + \Delta PlanCost\langle E^r, p^r, k\rangle) \\
&- \mathbf{min}_k PlanCost\langle E^r, p^r, k\rangle
\end{aligned}$$

The delta best cost of an OR node of the pair (expression E, property p) when E is a non-leaf level expression is:

$$\begin{aligned}
\Delta BestCost\langle E, p\rangle &= BestCost'\langle E, p\rangle - BestCost\langle E, p\rangle \\
&= \mathbf{min}_i PlanCost'\langle E, p, i\rangle - BestCost\langle E, p\rangle \\
&= \mathbf{min}_i(PlanCost\langle E, p, i\rangle + \Delta PlanCost\langle E, p, i\rangle) - BestCost\langle E, p\rangle \\
&= \mathbf{min}_i(LocalCost\langle E, p, i\rangle + BestCost\langle E^l, p^l\rangle + BestCost\langle E^r, p^r\rangle \\
&+ \Delta LocalCost\langle E, p, i\rangle + \Delta BestCost\langle E^l, p^l\rangle + \Delta BestCost\langle E^r, p^r\rangle) \\
&- BestCost\langle E, p\rangle
\end{aligned}$$

And when E is a leaf level expression: $\Delta BestCost\langle E, p\rangle = \Delta ScanCost\langle E, p\rangle$.

These definitions show that we can recursively define $\Delta PlanCost\langle E, p, i\rangle$ and $\Delta BestCost\langle E, p\rangle$ based on original values of *LocalCost*, *BestCost* and updates to the AND nodes $\Delta LocalCost$ or the leaf OR nodes $\Delta ScanCost$, and ultimately compute the output optimal tree T' of the re-optimizer with the least estimated cost, $\mathbf{min}_p BestCost'\langle E, p\rangle$.

Definition 12. Given an expression E, a query pre-optimizer computes an optimal plan tree for the expression, T. Given the state S within the pre-optimizer that we aim to keep for query re-optimization, and the updated costs or statistics U as the input to the re-optimizer program, an *incremental* query re-optimizer computes an updated optimal plan

T' for the original expression, as well as the new state S'. We call the policy of determining what intermediate state to be shared across re-optimizations to be the *memoization scheme*.

5.2 Procedural Incremental Re-optimization Algorithms and Optimizations: Bottom-up Style and Top-down Style

The declarative perspective on query optimization enabled us to define pruning strategies independently from execution flow, and with a clear definition of correctness. In fact, now that it is clear how the pruning strategies should work, we can "retrofit" them into both bottom-up and top-down-style query optimizers. More specifically, we can adapt the procedural query optimizer to incrementally recompute portions of a query plan (and the contents of the memoization table), and propagate control flow upwards or downwards as appropriate to propagate the effects of a change. Then a variety of our techniques from the prior sections can similarly be adapted:

- Incremental aggregate selection can be used to memoize suboptimal plans as well as optimal ones, and only propagate changes that affect the choice of a "best" alternative plan for a given expression.
- Incremental branch-and-bound can maintain a table of bounds for subplans, and this can be used as part of the pruning process.
- Separating the bounds computation enables it to be updated during enumeration both of alternative plans and superplans, as well as smaller subplans.

Now we present the procedural bottom-up and top-down style incremental re-optimization algorithms and their optimizations.

Bottom-up Style Re-optimizer In the bottom-up style query optimizer, one generally computes costs on the and-or-graph in the bottom-to-top level-by-level order with the use of dynamic programming. To adapt it to an incremental re-optimizer, we can start from computing $\Delta BestCost\langle E, p\rangle$ of OR nodes from the leaf-level, and recursively compute $\Delta PlanCost\langle E, p, i\rangle$ of AND nodes and $\Delta BestCost\langle E, p\rangle$ of OR nodes interleavingly.

The task of re-optimization finishes when we have computed $\mathbf{min}_p BestCost'\langle E, p\rangle$ to the original query execution E, the root level OR node.

We can trade-off memoization versus recomputation to reduce the cost of bookkeeping for suboptimal plans. On one extreme, if we memoize nothing during pre-optimization, then re-optimization will always need to do a full recomputation. On the other extreme, if we memoize everything, we might use up too much memory for bookkeeping. Hence in some cases, we can choose to only memoize the suboptimal plans most likely to become viable if small changes are made; and choose to discard the other alternatives, later recomputing them as necessary. This is a time and space complexity trade-off, but often a small amount of memoization can limit the recomputation to a few nodes in the and-or-graph. We can also cache the output when a re-computation occurs, to avoid multiple revisits of the same node during re-optimization. Here if we memoize all the old *BestCost* of OR nodes, as well as the old *PlanCost* of AND nodes, then we only need to compute $\Delta PlanCost\langle E, p, i\rangle$ of AND nodes and $\Delta BestCost\langle E, p\rangle$ of OR nodes based on the input delta costs and memoized costs. However, if we only memoize the old *BestCost* of OR nodes, but do not memoize the old *PlanCost* of AND nodes, then essentially we need to recompute them via their OR node children, $PlanCost\langle E, p, i\rangle = LocalCost\langle E, p, i\rangle + BestCost\langle E^l, p^l\rangle + BestCost\langle E^r, p^r\rangle$.

If we know in advance that the AND node who maintains the minimum cost value to its parent OR node *has not changed* or simply *decreased* its cost, then we only need to recompute those suboptimal sibling AND node costs that have been *decreased* because only those AND nodes might be the candidates for the new minimum cost. We call this heuristics to prune the recomputations as "Conditional Testing". Indeed, this idea leverages incremental aggregate selection from the declarative perspective, and later we will show in the experimental section that this strategy saves a lot of recomputations in practice.

Top-down Style Re-optimizer In the top-down style query optimizer, one generally computes costs on the and-or-graph in the top-down search order with the use of memoization and branch-and-bounding. To adapt it to an incremental re-optimizer, we need to consider how to compute $\mathbf{min}_p BestCost'\langle E, p\rangle$ to the original query execution E in a top-

down fashion, with special care to the delta changes to both the plan costs and bounds along the way. Since we know that an OR node of the pair (expression E, property p) and its entire subtree is pruned if and only if $BestCost\langle E, p\rangle > Bound\langle E, p\rangle$. Hence, both a change to the *BestCost* and the *Bound* could possibly change the result of this condition. Consequently, when this condition flaps, either a pruned subtree needs to be re-introduced, or an active subtree needs to be pruned.

Here we can leverage the idea from incremental branch-and-bounding from the declarative perspective, to understand how to incrementally maintain *Bound* as recursively defined by *LocalCost*, *BestCost*, $\Delta LocalCost$ and $\Delta ScanCost$ as well. Note that this computation can be performed in a different search order than the top-down search order, and different orders would result in different effectiveness of branch-and-bounding. Here we omit the discussions of the search order for computing *Bound* in a top-down procedural optimizer.

We can also apply the "Conditional Testing" optimization here in top-down style incremental re-optimization. Since now we not only have an aggregate on the cost of the OR node $BestCost\langle E, p\rangle = \mathbf{min}_i PlanCost\langle E, p, i\rangle$, but also have an aggregate on the maximum Bound on an OR node: $MaxBound\langle E, p\rangle = \max_i(ParentBound\langle E, p, i\rangle)$. Hence, we can leverage incremental aggregate selection from the declarative perspective, that if we know in advance that the AND node who maintains the maximum bound value to its parent OR node *has not changed* or simply *increased* its cost, then we only need to recompute those suboptimal sibling AND node costs that have been *increased*. Similar ideas to trade-off time and complexity with the memoizing best plans only scheme also apply here in the top-down search order.

Since both bottom-up and top-down style optimizations discussed here are procedural, they are subject to the original search order and data flow of their implementations. Hence, compared to a full declarative implementation, they lack the ability to be truly flexible in terms of search order and data flow, and hence is not easily extensible to parallel or distributed architectures.

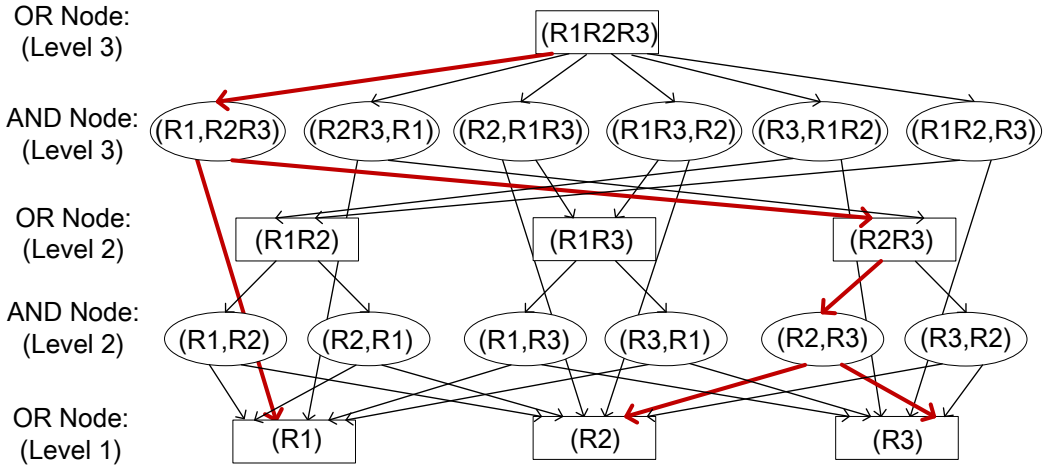


Figure 30: The and-or-graph for a simplified three way join query $R_1 \bowtie R_2 \bowtie R_3$. Red edges denote the best plan. Rectangles and ovals denote "OR" and "AND" nodes respectively.

5.3 Bounds of Recomputations for Incremental Re-optimization

In this section, we analyze the number of total AND and OR graphs for a simplified typical n way join query, and aim to understand the theoretical bounds for the necessary recomputations for incremental re-optimization should a cardinality of an OR node or a selectivity of an AND node changes.

Example 13. For a simplified query joining n relations, $R_1 \bowtie R_2 \bowtie \cdots R_n$. Suppose we only consider commutativity and associativity of joins as different ways of splitting a query, and we only consider one physical implementation of the join operator so that the logical and the physical operator tree is the same. And we do not consider the effects of pruning. See Figure 30 for the and-or-graph of a three way join query, where n=3. The leaf level OR nodes are called "level 1" OR nodes and they do not have any AND node children. The AND nodes and the OR nodes are interleaved: the children of a level k OR node are the level k AND nodes, and the children of a level k AND node are the level $k-1$ OR nodes. In this example, since n=3, the root node is a level 3 OR node.

At each level k, $1 \leq k \leq n$, there are exactly C_n^k OR nodes since we only consider commutativity and associativity of joins as different ways of splitting a query. Each of these OR nodes at level k has exactly $2^k - 2$ AND node children, also of level k. Now the

total number of AND nodes in this n-way join and-or-graph can be calculated as follows: at each level i, $2 \leq i \leq n$, since there are C_n^k OR nodes, and each has exactly $2^k - 2$ AND node children of level k. To sum them up, there are in total

$$\begin{aligned}
\sum_{i=2}^{n} C_n^i (2^i - 2) &= \sum_{i=2}^{n} C_n^i 2^i - 2 \sum_{i=2}^{n} C_n^i \\
&= \sum_{i=0}^{n} C_n^i 2^i - C_n^0 2^0 - C_n^1 2^1 - 2(2^n - C_n^0 - C_n^1) \\
&= (1+2)^n - 1 - 2n - 2^{n+1} + 2 + 2n \\
&= 3^n + 1 - 2^{n+1}
\end{aligned}$$

AND nodes in this typical simplified n-way join query. We also know that the total number of OR nodes can be calculated as follows: at each level i, $1 \leq i \leq n$, there are C_n^k OR nodes, hence to sum them up, there are in total $\sum_{i=1}^{n} C_n^i = 2^n - 1$ OR nodes in the and-or-graph of a typical simplified n-way join query. □

Hence, if we choose to memoize only the *BestCost* of OR nodes, we need to memoize $2^n - 1$ OR nodes; otherwise if we choose to memoize both the *BestCost* of OR nodes, and also the *PlanCost* of AND nodes, we need to memoize $2^n - 1$ OR nodes as well as $3^n + 1 - 2^{n+1}$ AND nodes. Suppose each signature occupies similar amount of memory, the ratio of memory consumption of memoizing all plans versus best plans only is: $(3^n - 2^n)/(2^n - 1)$. When n=5, this is around 7; when n=10, this is around 472.

Next, we discuss the theoretical bounds for the number of AND and OR nodes that need to be *updated* when there is a *cardinality* change of an OR node or a *selectivity* change of an AND node.

First, suppose there is a cardinality change on an OR node of level k ($1 \leq k \leq n$). We first analyze the number of OR nodes whose costs might be affected by this cardinality change. Since we know that only the super expressions, which contain the expression of this OR node might be affected by the cardinality change, and whose cost needs to be adjusted. We can easily compute that there are 2^{n-k} OR nodes whose corresponding super expressions contain the expression of the OR node of size k. Since we previously calculated that the total number of OR nodes for this n-way join query is $2^n - 1$. The ratio of *updated* OR nodes out of the total is $2^{n-k}/(2^n - 1) > (1/2)^k$. For example, if k=1,

then the update ratio of OR nodes is more than a half, meaning more than half of the OR nodes need to update their costs when there is a cardinality change on a leaf level OR node.

Now we calculate the number of AND nodes that need to be updated because of the cardinality change of an OR node of level k. At each level i above level k, $k < i \leq n$, there are C_{n-k}^{i-k} OR nodes that represent the super expressions of this expression, and each of these OR nodes has $2^i - 2$ AND children nodes, whose costs need to be adjusted. Hence, the total number of AND nodes whose costs need to be *updated* because of this cardinality change on an OR node of level k is:

$$\begin{aligned}
\sum_{i=k+1}^{n} C_{n-k}^{i-k}(2^i - 2) &= \sum_{i=1}^{n-k} C_{n-k}^{i} 2^{i+k} - 2\sum_{i=1}^{n-k} C_{n-k}^{i} \\
&= 2^k(\sum_{i=0}^{n-k} C_{n-k}^{i} 2^i - 1) - 2(\sum_{i=0}^{n-k} C_{n-k}^{i} - 1) \\
&= 2^k((1+2)^{n-k} - 1) - 2(2^{n-k} - 1) \\
&= 2^k 3^{n-k} - 2^k - 2^{n-k+1} + 2
\end{aligned}$$

Note that as shown previously, the total number of AND nodes in the and-or-graph for this n-way join query is $3^n + 1 - 2^{n+1}$. Hence, the update ratio of AND nodes out of the total number of AND nodes is $(2^k 3^{n-k} - 2^k - 2^{n-k+1} + 2)/(3^n + 1 - 2^{n+1}) > (2/3)^k$. When k=1, the update ratio of AND nodes is more than $2/3$, meaning more than $2/3$ of the AND nodes need to update their costs when there is a cardinality change on a leaf level OR node.

Second, suppose there is a selectivity change on an AND node of level k. It would certainly affect the cardinality of its parent OR node of level k, moreover, the children AND nodes of this parent OR node, in other words, the *sibling* AND nodes of this initial AND node, needs to update their costs as well. Hence, the update ratio of OR nodes out of the total number of OR nodes is still $2^{n-k}/(2^n - 1) > (1/2)^k$. The number of AND nodes that need to update their costs are: $2^k 3^{n-k} - 2^k - 2^{n-k+1} + 2 + 2^k - 2 = 2^k 3^{n-k} - 2^{n-k+1}$. Hence, the update ratio of AND nodes out of the total number of AND nodes is: $(2^k 3^{n-k} - 2^{n-k+1})/(3^n + 1 - 2^{n+1}) > (2/3)^k$.

Therefore, the conclusion for our analytical analysis of the bounds for recomputations in incremental re-optimization is: for a simplified n-way join query, if we only consider commutativity and associativity of joins as different ways of splitting a query, and we only assume one physical implementation for the join operator, without considering the effects of pruning, then a cardinality change of an OR node, or a selectivity change of an AND node, of level k, would result in at least $(1/2)^k$ of the total OR nodes to be updated, and at least $(2/3)^k$ of the total AND nodes to be updated. Later, we shall see in the experimental section the performance in practice.

5.4 Experimental Results

In this section, we discuss the performance of procedural-based incremental re-optimization. We applied the lessons learned from declarative incremental query optimization to traditional procedural-based optimizers, e.g., System-R style bottom-up optimizer with dynamic programming and Volcano-style top-down optimizer with branch-and-bounding. We implement a set of common functions (involving histograms, cost estimation and expression decomposition) for both top-down style and bottom-up style optimization; and we implement the search order and execution flow to mimic System-R style bottom-up style optimizer and Volcano-style top-down with branch-and-bounding. To enable incremental re-optimization, we leverage the ideas from Incremental Aggregate Selection and Incremental Branch-and-Bounding of declarative incremental re-optimization to make not only the plan costs and best costs but also the bounds adjustable. We also study the effects of different optimization techniques such as conditional testing, and different memoizing schemes on the performance of incremental-re-optimization.

In summary, we aim to answer the following questions empirically in this experimental section:

1. How different is the performance of procedural-based incremental re-optimization compared to their non-incremental counterparts?

2. How different is the performance of procedural-based incremental re-optimization between top-down style enumeration with branch-and-bounding versus bottom-up

Q5: SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'AMERICA' and o_orderdate $\geq$ '1993-01-01' and o_orderdate $<$ '1994-01-01' GROUPBY n_name;

Table 4: TPC-H Q5 benchmark query used in our experiments

style with dynamic programming?

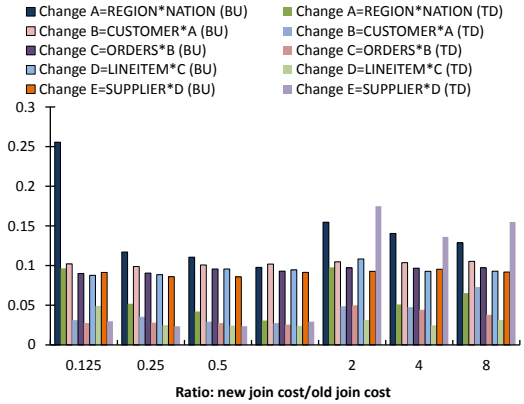
3. What are the effects of conditional testing, and different memoizing schemes on the performance of procedural-based incremental re-optimization?

4. Given the calculated theoretical bounds for the number of plan entries (OR nodes) and plan alternatives (AND nodes) that needs to be updated when a single *join selectivity* and a *scan cardinality* changes, how do they look in practice?

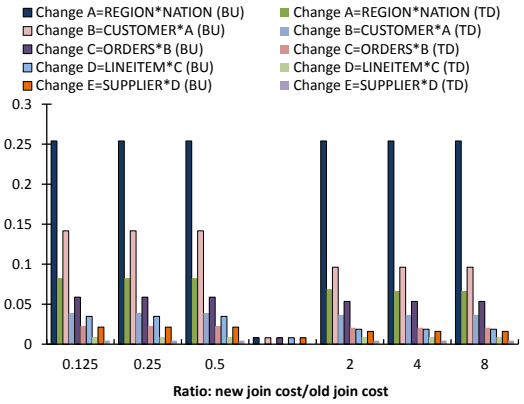
We focus on the TPC-H Q5 query (as shown in Table 4), a 6-way joins with aggregations query, throughout the experiments in this section, since we find that it exemplifies most of the discussions in this section. These experiments are conducted on a single local laptop machine with a dual-core Intel Core 2 2.53GHz with 4GB memory running 64-bit Windows 7 Professional. Performance results are averaged across 10 runs.

5.4.0.1 Comparison of Procedural-based Incremental Re-optimization between Bottom-up and Top-down Implementations, and versus their Non-incremental Counterparts

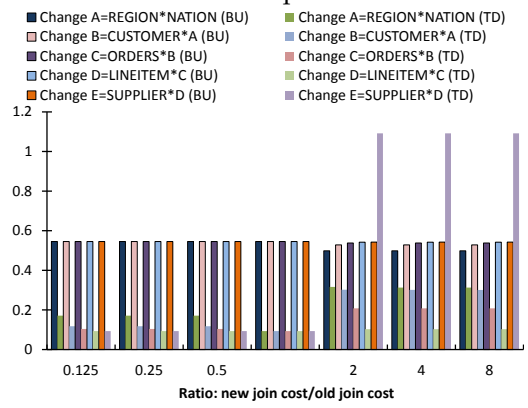
The first set of experiments shown in Figure 31 compares the performance of procedural incremental re-optimizer versus their non-incremental counterparts, in both top-down and bottom-up procedural implementations. Here we study what happens if a join's actual cost varies (by a factor ranging from 1/8th to 8x of its original cost) for TPC-H query Q5. Here we memoize not only the best-plans, but also alternative subplans as well during re-optimization. Figure 31 a) shows the running time for incremental query re-optimization (normalized to the time for non-incremental counterparts), varying different join's cost changes. The "(BU)" entries are bottom-up implementations,



(a) Optimization time ratio to non-incremental counterparts



(b) Recompute ratio: plan table entries (OR nodes)



(c) Recompute ratio: plan alternatives (AND nodes)

Figure 31: Performance of top-down vs bottom-up-style procedural incremental re-optimization of TPC-H Q5 — upon change to a join cost value

whereas "(TD)" references a top-down implementation with branch-and-bounding. Observe that in general incremental re-optimization is 4-30 times faster than a complete non-incremental optimization, in both bottom-up and top-down implementations. As expected, the results validate that top-down strategies are generally superior to bottom-up ones for incremental re-optimization, because they enable greater amounts of pruning which results in less work in incremental re-optimization. The only exception to this is the last data point, where we change the join cost of $E = Supplier \bowtie D$: here, increasing this join cost causes the top-down re-optimizer to take more time on incremental re-optimizer than the bottom-up implementation, possibly because the previously pruned nodes are re-introduced more often in this case.

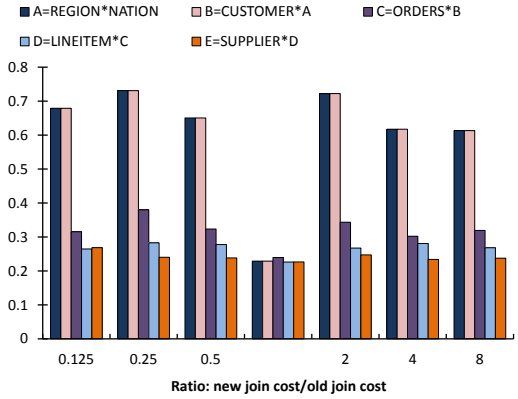
We can see this in more detail in Figure 31 b), which shows the proportion of recomputed plan table entries (OR nodes), and c), which shows the proportion of recomputed plan alternatives (AND nodes). In general, the larger the subexpression for which a join cost changes, the less work is required to re-compute, and the more benefit is provided by incremental re-optimization. From Figure b), we can see that the higher level an AND node is associated with whose join cost has been changed, the less number of OR nodes it needs to recompute, and generally the recompute ratio of OR nodes is below 30%. The number of recomputed AND nodes (in Figure c) shows that bottom-up re-optimizers incur constantly below 60% recomputations of the total AND nodes, whereas the top-down re-optimizers incur far less recomputations of AND nodes (below 20%) when a join cost decreases and more recomputations when a join cost increases (still below 120%), mainly due to re-introducing AND nodes that were previously pruned.

5.4.0.2 Effects of Conditional Testing

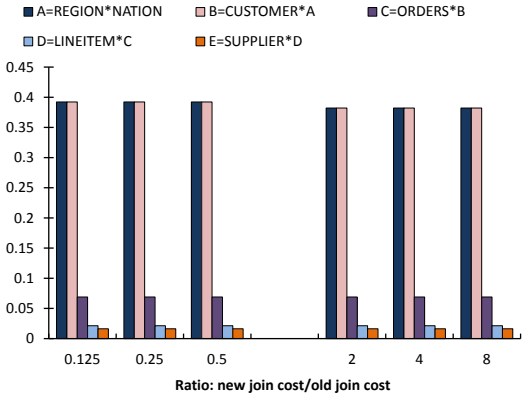
Figure 32 shows the effects of conditional testing on bottom-up incremental re-optimization. We study the performance ratio of applying the conditional testing versus non-optimized version, where we memoize best plans only in bottom-up style incremental re-optimization. We can see from Figure a) that the conditional testing scheme in general brings about 1.3 to 4 times of speedup in terms of optimization time. We can see more detail from Figure b) that the conditional testing scheme recomputes around 2% to 40% of the OR nodes versus a non-optimized version, and from Figure c) that the conditional testing scheme recomputes around 2% to 45% of the AND nodes versus a non-optimized version. Overall conditional testing brings a huge speedup to incremental re-optimization.

5.4.0.3 Comparisons Between Memoizing Best Plans and Memoizing All Plans

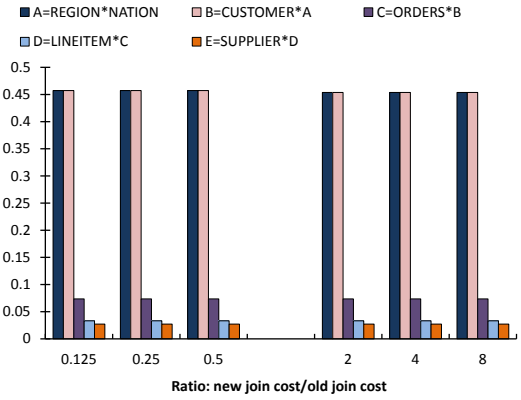
The set of experiments shown in Figure 33 measures the performance difference between memoizing best plans (OR nodes) versus memoizing all plans (AND nodes) during top-down style incremental re-optimization. Here we study what happens if a join's actual cost varies (by a factor ranging from 1/8th to 8x of its original cost) for TPC-H query Q5. The "(Best)" entries denote memoizing only the best plans (OR nodes), whereas



(a) Optimization time ratio to non-optimized version



(b) Recompute ratio: plan table entires (OR nodes)

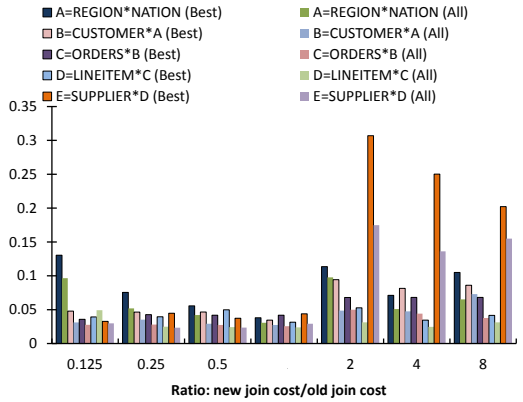


(c) Recompute ratio: plan alternatives (AND nodes)

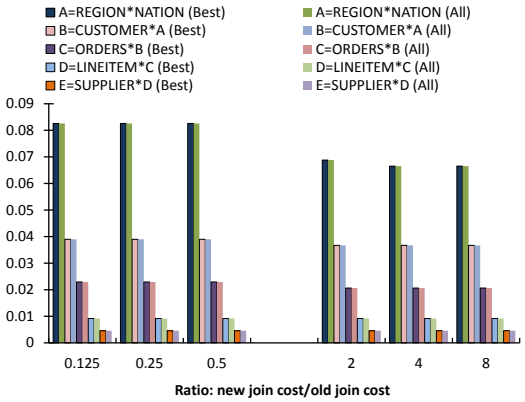
Figure 32: Performance ratio of conditional testing vs non-optimized version during bottom-up style incremental re-optimization of TPC-H Q5 — upon change to a join cost value

"(All)" entires denote memoizing all the alternative plans as well (both AND and OR odes). Figure 33 a) shows the difference of *execution time* between memoizing best plans versus memoizing all plans, and the absolute values show the proportion to their non-incremental counterparts. We can see that the "Best" scheme brings about 3-35 times speedup compared to its non-incremental version, and the "All" scheme brings about 6-35 times speedup compared to its non-incremental version. In general the "All" scheme is faster than the "Best" scheme for top-down incremental re-optimization, but in this TPC-H Q5 query less than 2x faster.

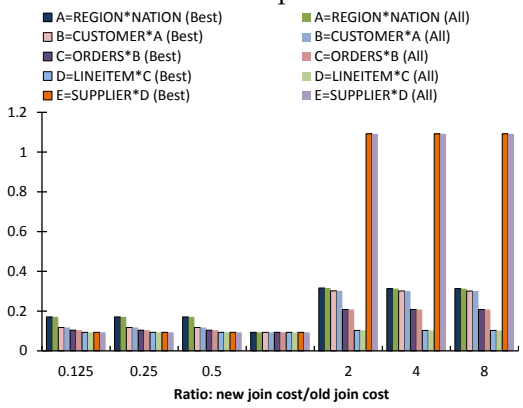
Figure 33 b) shows the proportion of recomputed plan table entries (OR nodes) and c) shows the proportion of recomputed plan alternatives (AND nodes) during incremen-



(a) Optimization time ratio to non-incremental counterparts



(b) Recompute ratio: plan table entries (OR nodes)



(c) Recompute ratio: plan alternatives (AND nodes)

Figure 33: Performance of memoizing best plans vs memoizing all plans during top-down style incremental re-optimization of TPC-H Q5 — upon change to join cost value

tal re-optimization. Both figures show that the "Best" scheme and the "All" scheme recompute the exact same number of AND and OR nodes. In general, top-down style procedural re-optimization incurs less than 10% of the total OR nodes and less than 120% of the total AND nodes.

5.4.0.4 Changing a Single Join Selectivity or a Scan Cardinality

Figure 34 demonstrates the performance of bottom-up incremental re-optimization when an AND node changes its join selectivity, or a leaf level OR node changes its scan cardinality. Here we memoize all the alternative plans as well as the best plans. On the left side, Figure a), c), and e) show the normalized execution time ratio, recompute ratio of AND

nodes, and recompute ratio of OR nodes, respectively, when a *join selectivity* varies 1/8th to 8x of its original value. On the right side, Figure b), d) and f) show the normalized execution time ratio, recompute ratio of AND nodes, and recompute ratio of OR nodes, respectively, when a *scan cardinality* varies 1/8th to 8x of its original value. Since TPC-H Q5 query is a 6-way join query, there are in total 6 scan relations, and they all represent level 1 OR nodes.

From Figure a), we can see that when an AND node changes its join selectivity, incremental re-optimization generally brings about 6-10 times of speedup compared to their non-incremental counterpart. The higher level the AND node of the selectivity change is, the bigger the speedup we see. From Figure b), we can see that when an OR node changes it scan cardinality, incremental re-optimization generally brings about 3-10 times of speedup compared to their non-incremental counterpart.

Figure c) and d) show the recompute ratio of AND nodes during incremental re-optimization, when an AND node changes its join selectivity, or an OR node changes its scan cardinality, respectively. Recall that our theoretical lower bounds for recomputed AND nodes for a typical n way join is $(2/3)^k$ where k is the level of the AND node whose selectivity has been changed, or the level of the OR node whose cardinality has been changed. Here we can see that in both situations, the recompute ratio of AND nodes are constantly between 45% to 55%. Note that since the TPC-H query Q5 not only includes joins but also aggregates, and we consider several physical join implementations instead, there might be slight difference in the computations. Generally, this matches then bound where changing AND node's selectivity ($k > 1$) and empirically show smaller number of updates of AND nodes when changing a scan cardinality (k=1).

Figure e) and f) show the recompute ratio of OR nodes during incremental re-optimization, when an AND node changes its join selectivity, and an OR node changes its scan cardinality, respectively. Recall that our theoretical lower bounds for recomputed AND nodes for a typical n way join is $(1/2)^k$ where k is the level of the AND node whose selectivity has been changed, or the level of the OR node whose cardinality has been changed. Here we can see that in the case of a join selectivity change, the actual recompute ratio of OR nodes in practice matches our theoretical analysis very well: for a join selectivity change on an AND node of level 2, we see the recompute of OR nodes around 0.25, and then the

ratio gets cut half when the level of AND nodes increases. In the case of a scan cardinality change, we can see around 25% to 65% of the recompute ratio of OR nodes, whereas the theoretical lower bound is 50%.

5.4.0.5 Summary of Experimental Results

Here we summarize the experimental results in this section:

1. In general procedural incremental re-optimization is 4-30 times faster than a complete non-incremental optimization.

2. Top-down strategies are generally superior to bottom-up ones for incremental re-optimization, because they enable greater amounts of pruning hence result in less work in incremental re-optimization.

3. Conditional testing brings a 1.3x-4x speedup of optimization time to incremental re-optimization.

4. Memoizing best plans usually incurs more optimization time than the memoizing all plans scheme, but no more than 2x for our top-down implementation of the incremental re-optimizer for TPC-H Q5 query.

5. We see in practice around 45% to 55% of recomputed AND nodes when a join selectivity or a scan cardinality changes, 25% to 65% of recomputed OR nodes when a scan cardinality changes, and around $(1/2)^k$ of recomputed OR nodes when a join selectivity of a level k AND node changes. These results generally follow the theoretical bounds, and the slight difference might be due to our more complex query (e.g., involving not only joins but also aggregates) and more physical operators per logical operator compared to a more simplified assumption in the theoretical model.

5.5 Conclusion

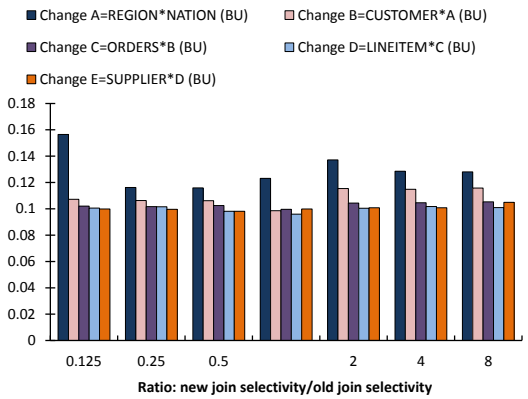
In this section, we studied incremental query re-optimization approaches for full-fledged cost-based procedural query optimization frameworks. We make the following contributions in this work.

- We define the cost-based procedural incremental query re-optimization problem in a formal way.

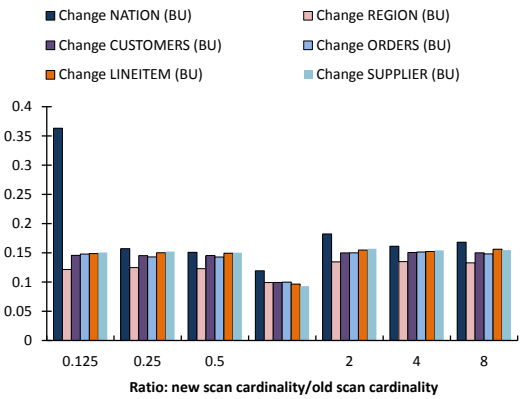
- We leverage the lessons learned from the declarative perspective and present procedural incremental re-optimization algorithms for both bottom-up and top-down style architectures. We also study different optimizations to minimize recomputation and book-keeping: conditional testing, and memoizing best plans only.

- We analyze the worst-case bounds for re-computations of AND and OR nodes during incremental re-optimization. We show that for a simplified n-way join query, if we only consider commutativity and associativity of joins as different ways of splitting a query, and we only assume one physical implementation for the join operator, without considering the effects of pruning, then a cardinality change of an OR node, or a selectivity change of an AND node, of level k, would result in at least $(1/2)^k$ of the total OR nodes to be updated, and at least $(2/3)^k$ of the total AND nodes to be updated.

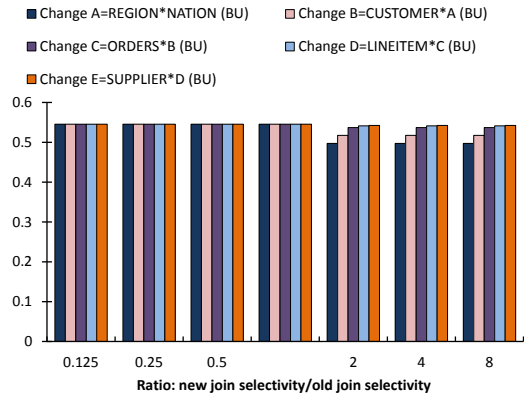
- We empirically evaluate our procedural incremental re-optimization algorithms and study the performance differences between top-down and bottom-up style incremental re-optimization, and the difference to their non-incremental counterparts. Results show that our procedural incremental re-optimization is 4-30 times faster than a complete non-incremental optimization, and top-down strategies are generally superior to bottom-up ones for incremental re-optimization. We study the effects of different optimizations, and see that conditional testing brings about 1.3x-4x speedup in re-optimization time, and memoizing best plans saves memory consumption by not exceeding 2x optimization time compared to the memoizing all plan scheme. Finally, we see in practice the actual number of recomputed AND and OR nodes during incremental re-optimization, and compare them against the theoretical bounds we presented earlier.



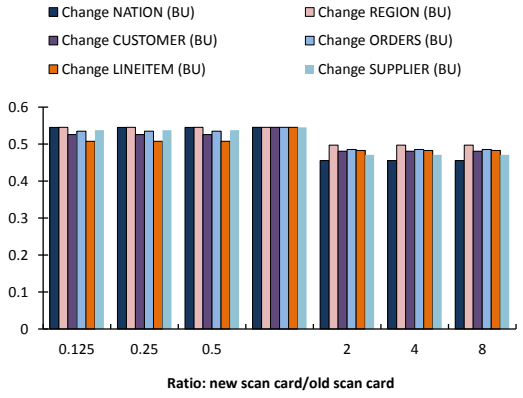
(a) Optimization time ratio to non-incremental when changing a join selectivity



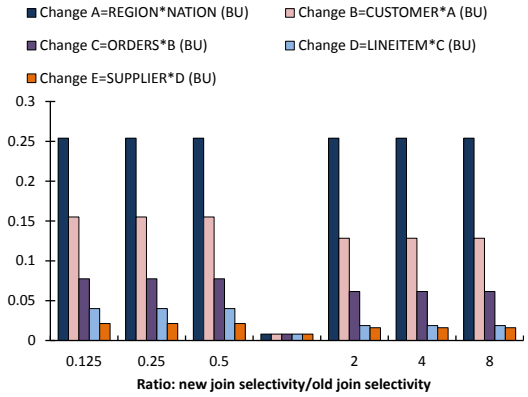
(b) Optimization time ratio to non-incremental when changing a scan cardinality



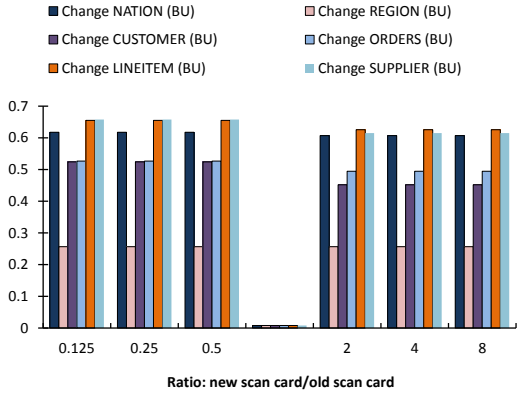
(c) Recompute ratio: AND nodes when changing a join selectivity



(d) Recompute ratio: AND nodes when changing a scan cardinality



(e) Recompute ratio: OR nodes when changing a join selectivity



(f) Recompute ratio: OR nodes when changing a scan cardinality

Figure 34: Normalized number of recomputed AND nodes to non-incremental bottom-up

Chapter 6

A Case Study: The Aspen System

In this chapter, we describe the design and implementation of our prototype stream processing system Aspen, as well as a case study on a real-life application scenario SmartCIS. We start this chapter with general motivations of developing Aspen and SmartCIS.

Low-cost networked sensors are resulting in a new class of applications that combine data from the "digital world" with sensor readings, to create environments that intelligently manage resources and assist humans. Examples include intelligent power grids [91], smart hospitals [90], home health monitors, energy-efficient data centers, and building visitor guides. In such applications, there is a need to bring together disparate data from databases (e.g., site information, patient treatments, maps) with data from the Web (e.g., weather forecasts, calendars), from streaming data sources (e.g., resource consumption within a server), and from sensors embedded within an environment (e.g., generator temperature, RFID readings, energy levels) — in order to support decision making by high-level application logic. Today this sort of data integration, if done at all, is performed by a proprietary software stack over fixed devices.

In order for intelligent environments to reach their potential, what is necessary is an *extensible*, *multi-purpose* data acquisition and integration substrate through which the application can acquire data — without having to be coded with special support for new device or network types. Over the past 30 years, the database community has developed a wealth of techniques for performing data integration through views and related formalisms [57]. Likewise, declarative queries have been shown to be useful beyond

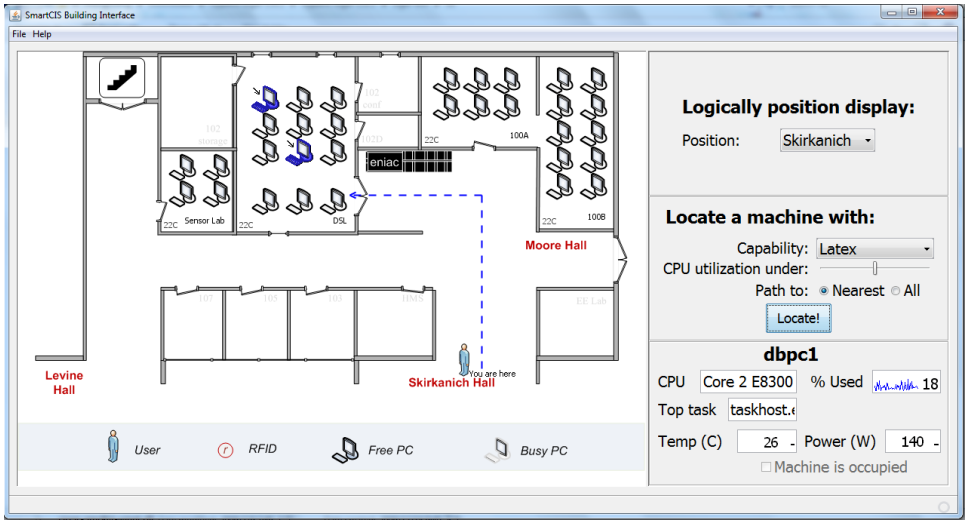


Figure 35: Display indicating a path to, and information about, the nearest machine with LaTeX.

databases, with extensions for distributed data stream management [8, 14, 24, 51] and sensor networks [32, 36, 71]. The key question is how to develop a unified declarative query and integration substrate, which supports a multitude of stream and static data sources on heterogeneous, possibly unreliable networks. Computation should be expressed in a single query language and "pushed" to where it is most appropriate, taking into account capabilities, battery life, rates of change, and network bandwidth.

The ASPEN (Abstraction-based Sensor Programming ENvironment) project [62, 64] that we have developed at Penn tackles these issues, extending the formalisms of data integration (schema mappings, views, queries) to the distributed stream world. The novelty of our solutions to prior work include 1) new query processing algorithms suitable for integrating highly distributed stream data sources, both in low-power sensor devices [75, 76] and more traditional PCs and servers [66, 67], 2) query optimization techniques for federations of stream processors specialized for sensor, wide area, and LAN settings, and 3) new datatypes, query extensions, and data description language abstractions for environmental monitoring and for routing information to users. In support of smart environments, we seek a *single data access layer* for integrating sensor, stream, and database data, regardless of origins.

The showcase application for the ASPEN architecture, which we term SMARTCIS, in-

volves instrumenting Penn's Computer and Information Science (CIS) Department buildings, labs, and data centers to help improve energy efficiency, guide visitors to their desired destinations, and locate resources. Our live demonstration of SmartCIS [62] at SIGMOD 2009 received Honorable Mention for Best Demo. SmartCIS consists of GUI and query logic built over the Aspen data integration substrate. It combines information from on-site sensors (e.g., pressure-sensitive seat cushions, RFID tags, energy meters) with data from the Web (calendars) and from our Distributed Systems Laboratory at Penn (machine and desk-occupied status; machine configurations).

In this chapter, we will demonstrate 1) the architectural design of a declarative smart environment system; 2) how Aspen enables a *uniform* stream acquisition framework for various data sources such as physical sensors, digital streams, web data as well as databases; and finally, 3) how Aspen uses a *federated query optimizer* that is able to partition a query specified over heterogeneous data sources into a series of subqueries defined over a specific type of data sources. We first describe the SmartCIS as an example application of our Aspen system in Section 6.1. Then we present the underlying Aspen system: its architecture (Section 6.2), federated query optimizer (Section 6.3), and stream query processor (Section 6.4). Finally, we conclude in Section 6.5.

6.1 SmartCIS Building Application

One of the most compelling emerging applications of sensors are intelligent building environments: they promise to make the experience of visiting a large building or a hospital less disorienting, to make buildings or large data centers more energy-efficient, to help occupants remember to take their medications or make it to a next meeting. A distinguishing feature of such environments, versus other sensor network applications, is a need to bring together database data with streaming data from the Web or Internet and streaming data from sensor devices. The task of designing a smart building usually can be separated into three tiers: data acquisition and integration, query and control logic, and a user-interface view (analogous to model-view-controller architectures).

Our Aspen/SmartCIS system focuses on monitoring and querying the data of interest to CIS students and faculties, as well as system administrators: lab status, machine

activity, resource consumption, and machine physical state. We target two main tasks: giving a real-time update of the building state, and guiding students to the resources they need. Through the SmartCIS GUI, visitors can see occupied and unoccupied desks in the laboratories and on-site (detected through the seat sensors); their positions in the building (obtained via RFID); temperature, light, and energy usage levels for every machine and lab; room reservation status from Google Calendar; and the resources available at each machine (e.g., software, special equipment). Visitors can see status information or issue a query for directions (a physical path) to a machine with a particular resource.

6.1.1 User Experience

SmartCIS interacts with users through a touch interface on a kiosk or (for the demo) a tablet PC. Figure 35 shows a screen shot of our graphical interface, which centers around a building schematic. In the full application, the user will see the individual information on a kiosk located somewhere in the building. Our screen shot shows the demo application, which has a selector in the upper right-hand corner enabling a SIGMOD attendee to choose a simulated kiosk location.

Buildings, entrances and exits, rooms, and machines are illustrated schematically. Their status is refreshed in real-time based on data streams from the environment and the Web, combined with database information about locations and configurations. Rooms are grayed out when marked as reserved in a standard Google calendar, or when their lights are out (as detected by sensors). Machines are grayed out when they are currently in use (as detected by high CPU utilization or a pressure-sensitive seat cushion connected to a Crossbow iMote). The presence of a user is detected through active RFID tags (IRIS motes that broadcast a low-power signal that is tracked by stationary motes located throughout the building hallways) and is indicated in the schematic.

The user can also trigger new continuous queries over the streaming data in the system. Clicking on a machine icon switches the right-hand pane to show details about that device: its host name (from a database table mapping coordinates to machine identities), CPU type (also from the database), CPU utilization and the most CPU-intensive task (from a "soft sensor" application), temperature (from an iMote), and energy (from a

USB energy meter or an IP-based Power Distribution Unit or PDU). A double-click opens up a secondary window showing energy consumption on a per-task basis (scaling overall energy consumption by the amount of resources consumed by each process). Finally, a visitor can also request to be directed to an available machine with specific resources (e.g., software packages like Microsoft Office or a video editor). A shortest-path query is initiated between the user's current location and the nearest available room with the specified resource.

6.1.2 Sensors and Data Sources

The data sources underpinning SmartCIS are heterogeneous, requiring a variety of *wrappers* (interface modules), and can be divided into four broad categories.

Sensor devices. We use Crossbow IRIS and iMote2 sensors to monitor the rooms' and workstations' temperatures, as well as light levels (useful for determining if a lab is open). A pressure-sensitive seat cushion attached to a wireless mote monitors whether someone is seated at each desk in the lab. A "wrapper" periodically extracts this value and sends it along a data stream. Energy meters are physically plugged into machines and feed raw readings into the system. To track users' locations, "mote" sensors are embedded in the hallways at major intersection points, at approximately every 50 feet. These sensors listen for a "beacon" transmission from an active RFID device (also a mote) carried by an occupant and based on the strength of the signal determine where that person is positioned in the building.

"Soft" sensors. Servers and workstations run daemon softwares to monitor machine activities: job executions, users logged in, CPU utilizations, number of requests being handled in a Web server application, etc. In addition, the status of Aspen, our back-end data acquisition and integration substrate itself, is also monitored: the queries and plans being executed, the counts of tuples received and sent for every operator, etc. This helps developers diagnose problems at the query execution level and also helps determine per-query energy usage.

Web and streaming data sources. A wrapper periodically polls a Google Calendar for room reservations. Another wrapper polls energy usage from a Web interface to our lab's

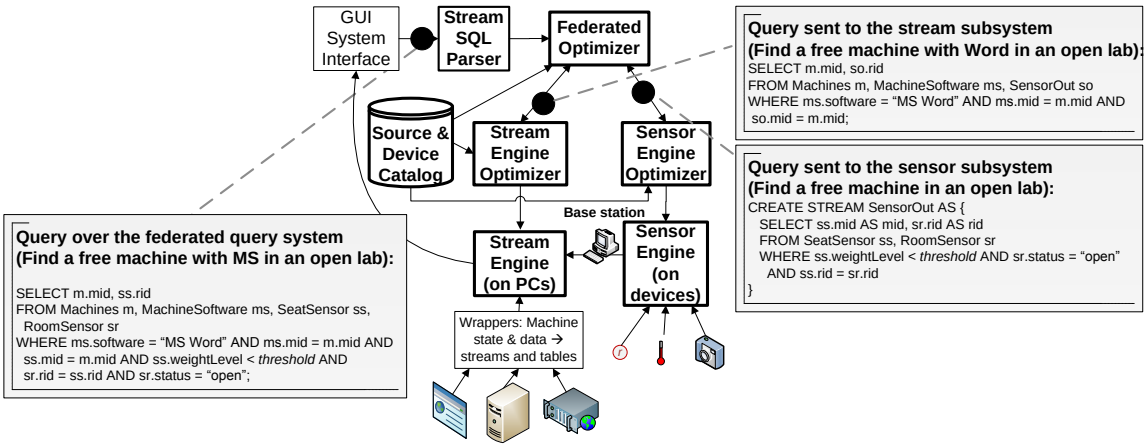


Figure 36: Architecture of SMARTCIS, including ASPEN components in bold.

power distribution units (PDUs).

Databases. A conventional DBMS stores the coordinates of each RFID detector (the motes have no built-in absolute positioning capability), a list of machine configurations and locations, and a table of "routing points" describing possible path segments and distances in the building in order to suggest routes to resources.

One of the novelty of our system is that the data streams from these inputs are "hooked" to the SMARTCIS GUI through a series of Stream SQL queries [87] and view definitions, plus callbacks to Java functions that update the graphical widgets. They are done through different implementations of a uniform abstraction wrapper. On top of this architecture, It is trivial to extend the GUI to support visual or auditory alarms if machines exceed a temperature or load factor, or to aggregate the sensor data across users, applications, or machines. Even the path routing in the GUI can be done declaratively, using our recursive extensions to Stream SQL and support using the techniques we introduced in Chapter 3. We next describe how SMARTCIS maps onto the ASPEN substrate that provides distributed Stream SQL services.

6.2 System Architecture

The SmartCIS system consists of three major components: the graphical interface described previously, which can be deployed on kiosks; the Aspen data integration and acquisition substrate, which includes two query runtime systems (one that enables certain computations to be "pushed" to sensor devices, and one that does distributed stream processing over PC-style servers) plus a federated query optimizer; and wrappers and interfaces over the actual sensors, databases, and machines. (See Figure 36.) Components of the Aspen substrate appear in boldface. (For full data stream integration Aspen will also include support for schema mappings and query reformulation, but SmartCIS does not require these components.)

Most of the research innovations are in the Aspen modules. Aspen takes a query (Stream SQL with extensions for devices and for routing query output to displays) and invokes a federated query optimizer that partitions it into two portions (see Figure 36): a subquery that is "pushed" out to the sensor network and sensor devices, and the remaining computations that get executed on our distributed stream engine for servers.

The distributed sensor engine, whose core features were described in [75], is novel in supporting not only aggregation and selection queries over sensor devices, but also *in-network* joins between devices. This is useful in SmartCIS, for instance, when we return machine temperature data for workstations that are in use. We detect that a workstation is being used by checking the status of seat cushions as well as the light level at an adjacent chair. The most efficient query strategy is to perform a proximity-based join between status of seat cushion and light level sensors (with a threshold applied on the light level), and route the temperature information across the sensor network only if the light level threshold is not met. A query optimizer decides where to perform the join computation on a sensor-by-sensor basis.

Our distributed stream engine, described in [66, 67, 92], supports not only Stream SQL queries over windowed data, but also *transitive closure queries* to compute neighborhoods and paths. The stream engine brings together streaming data, database data, and the data returned by the subqueries sent to the sensor engine. It is also responsible for computing suggested routes for building occupants to get to their destination: this can be done in

real-time based on the occupant's current position and information about the topology of the buildings (connected by routing points described previously).

6.3 Federated Optimizer

As discussed in the previous section, one of the major components of ASPEN is its federated query optimizer. The job of a federated query optimizer is to find the optimal query plan over federated data sources, which may span across multiple subsystems, each of which has its own custom optimizer and cost metric, customized to the target device and network capabilities (e.g., energy, latency, bandwidth). We give an example of the task of the federated optimizer in SMARTCIS.

Example 14. Suppose we have two types of sensors deployed in the lab, seat sensors and room sensors. Each seat sensor is pre-initialized with information about its position relative to a machine and the room; it reports the occupied-status of the seat cushion to which it is attached. Each room sensor is pre-initialized with its room, and detects the current light level to tell whether the room is occupied or not. Suppose we also have a static table *Machines* storing machine information for the lab, and a dynamic stream *MachineSoftware* containing information about installed software and versions from a web page. The user may pose a query to find all the free machines in an open lab which have the software "Word".

```
SELECT m.mid, sr.rid
FROM Machines m, MachineSoftware ms, SeatSensor ss, RoomSensor sr
WHERE software = "Word" AND ms.mid = m.mid AND ss.mid = m.mid
  AND ss.weightLevel < threshold AND sr.rid = ss.rid AND
  sr.status = "open";
```

□

There are multiple plausible ways of splitting the query for subsystems (both sensor and stream) to execute. One method pushes the *SeatSensor-RoomSensor* join and all relevant selection conditions to the sensor subsystem, then sends the output to the stream engine. (Example SQL for this scenario is shown in Figure 36.) Alternatively, we can issue *two* subqueries to the sensor subsystem: one to fetch *SeatSensor* readings above threshold, and the other to fetch *RoomSensor* readings with open status. The join between the two

will be done at the stream end. Intuitively, the first query partitioning is likely to return fewer results to the stream system only if the predicates are selective.

The search space of candidate plans for a federated query optimizer is quite huge. First, combing query fragments can be realized through a bunch of join methods: two-way join (semi-join, bloom-join, natural-join, or cartesian products) and multi-way join, etc. These join methods might be executed in a decentralized fashion, thus we also need to explore all possible ordering of joins and parallelism opportunities, as well as all potential locations (such as sending join predicates to the base station, all nodes in the sensor network, grouped nodes, horizontally partitioned nodes, etc) to execute the joins.

The federated optimizer must choose among these candidate plans by minimizing an over-arching cost metric (e.g., query latency). This metric may be *different* from the metrics of the "local" optimizers for the underlying stream and sensor engines (e.g., bandwidth, energy consumption). The federated optimizer must find a query partitioning that, when each subquery is optimized according to its target platform's specific metric, results in the best plan with respect to the federated optimizer's over-arching metric. Its plan enumeration strategy resembles that of [48], which predicts the query plan produced by an external optimizer, in order to produce the minimum-cost plan according to its own metric.

We have developed heuristics to prune the huge search space of candidate plans. For example, we partition a query involving both sensor and stream sources into subqueries, where we limit the number of in-network joins performed at a sensor subsystem to only *one*. This means a join between sensor sources could either be done in-network, at the base station or at the stream engine, and if done in-network, the results must be sent to the base station. We exploit the opportunities for semi-join (collecting unique values of the joined attributed of one relation to be probed against another), bloom-join (building bloom-filters on one relation) as well as natural joins and we assume cartesian products are usually expensive hence could not be done in-network. Query partitioning itself is a non-trivial task as it needs to maintain the correct algebraic transformations of SPJ queries while exploring all possible candidate subquery combinations.

6.4 Stream Engine

Our stream engine is derived from the distributed SQL query processing engine from ORCHESTRA [92]. This engine supports horizontal partitioning of data across nodes within a cluster or peer-to-peer network, and is based on a push-style query processing model. It supports full SPJ queries, as well as unions and aggregates. We enhance the engine with support for continuous queries (where the query is active unless deliberately stopped) over windows, where the size of the sliding window tells the system when to evict expired tuples. The engine can seamlessly combine data from streaming sources, tables partitioned throughout the cluster, ODBC/JDBC sources, and the sensor query engine. The query optimizer uses a Volcano-style top-down plan enumeration and branch-and-bound algorithm, and takes into account the network latency as well as data transmission rate when estimating the cost of a certain query plan.

A novel aspect of our engine is its support for *transitive closure* queries (such as shortest paths) computed (and incrementally maintained) over streaming data. Such queries commonly appear in sensor settings. We have developed techniques, discussion in Chapter 3, based on (1) the use of a particular kind of *data provenance* that enables us to detect when a tuple in the output stream should be expired, (2) early pruning of intermediate results that do not contribute to the output, and (3) careful use of buffering to reduce traffic. SMARTCIS exploits these features to compute path queries, when a user requests to be directed to a resource within the building.

6.5 Conclusion

This chapter provided a case study of the SMARTCIS "smart building" application and the design and implementation of its underlying ASPEN prototype system. We introduced new query processing schemes for integrating highly distributed stream data sources, as well as query optimization techniques for federations of stream processors.

Chapter 7

Related Work

In this chapter, we survey several important lines of related work for this dissertation. Generally, our related work falls into two categories: related systems (e.g., data stream management systems, rule-based active database systems, deductive database systems, sensor query processing systems, map-reduce systems, and adaptive query processing systems) and related approaches (e.g., other provenance alternatives, declarative approaches, and incremental view maintenance approaches). Below we review each one in detail.

Data stream management systems. In the past decade, several influential data stream management systems (DSMS) [14, 24, 51, 77, 88] have been proposed. These systems have established the basic semantics and query languages for stream processing. In parallel, stream SQL techniques have shown to be highly advantageous in sensor settings [32, 36, 71]. Work such as REED [2] has shown that there is great promise in coupling these two classes of systems. To the best of our knowledge, none of these systems support recursive queries natively in the execution engine. We share many similar goals with data stream management systems [14, 24, 51, 77, 88] and complex event processing systems [20, 46], such as real-time delivery of results, smart memory management mechanisms, extended unified stream and non-stream query languages, and so forth. Indeed, in this dissertation, we leverage many important language- and architectural designs from those systems, such as push-based query processing, stream data models, and stream query languages. Our focus in this thesis is on extending capabilities of data stream management systems to answering recursive queries natively, and support for incremental query re-optimization

to improve the overall efficiency of data stream processing.

Rule-based active database systems. A typical active database system [99] features an event driven architecture (often in the form of Event-Condition-Action rules) which can respond to conditions both inside and outside the database. Indeed, active database systems are among the first to provide monitoring and triggering capabilities in database management systems. In ECA rules, triggered events usually include data modifications, data retrieval, or temporal events; conditions usually refer to database predicates and database queries; and actions usually involve data modifications, data retrieval or application procedures. Omitting the event part would be more declarative, but it sacrifices the flexibility that different actions can be specified when a condition is satisfied depending on which event occurred. The main difference between our work and active database systems is the level of declarativity of the language and the system architecture. ECA rules are known to be less declarative than *datalog* rules, hence they involve more physical details such as data modifications or the order of executing rules. Performance-wise active database systems are not as successful as commercial DBMSs, however, active database systems influence many subsequent systems such as stream query processing systems and complex event processing systems. We leverage the ideas of monitoring and triggering events from active database systems, however, we use high-level declarative languages to specify events and actions, such as computing shortest paths. Overall our architectural design and fundamental assumptions are more similar to data stream management systems.

Deductive database systems. Rules also form the core of deductive database systems. Indeed, declarative logic programming style rules, e.g., in *datalog*, are used in deductive database systems to add the power of recursively defined views to conventional DBMSs [99]. Most work in this area focus on developing efficient strategies for queries on recursively defined views. The techniques related to this dissertation are deductive logic programming techniques to address general incremental view maintenance and optimization problems over recursively defined views, such as semi-naïve evaluation [15], magic sets [16] and aggregate selection [89]. We extend many ideas from this field in this dissertation.

Sensor query processing systems. In support of smart environments, we seek a single data access layer for integrating sensor, stream, and database data, regardless of origins. This single programming interface over heterogeneous sensors and stream sources distinguishes us from other database-style sensor systems [32, 36, 71], which focuses on specialized sensor query processing engines. We enable generalized declarative queries that integrate both sensor data and stream data, and introduce stream-based techniques that were not supported for sensor query processing systems.

Map-Reduce and Spark systems. Recently there is a proliferation of large-scale distributed engines built on top of the Map-Reduce and Spark paradigm [31, 101]. Map-Reduce was originally developed for analyzing unstructured data, such as Web documents and Web search query logs, on large-scale shared-nothing commodity servers. Recently, such engines are extended to support complex SQL-like queries on top of the lower-level Map-Reduce operations [93, 100] essential for traditional enterprise data warehousing use cases. Spark is a more recent distributed computing paradigm that utilizes in-memory RDDs (Resilient Distributed Datasets) for intermediate steps of multi-stage transformations, hence can more efficiently compute batch queries. In comparison to Map-Reduce and Spark systems, we support data models that are not just static, but also dynamic, and our execution models can support stream-based query executions, such as stream joins and stream aggregates, and so forth, more in the flavor of data stream management systems. Also, our work on recursive query processing and adaptive query processing may advance the understanding of supporting recursion and incremental query re-optimization over Map-Reduce and Spark-based systems.

Adaptive query processing systems. Adaptive query processing has been extensively studied in the 1990s. Initial efforts include query re-optimization at materialized points [56], CPU scheduling-based adaptation [50, 94] and redundantly-computed adapting methods [6]. Recently there has been work in addressing the moving state problem [54, 72, 85]. Apart from these plan-based methods, a tuple-based routing scheme, Eddies [10, 83], has been proposed to re-route tuples to plans in highly dynamic environments. There are also a few surveys that summarize various aspects of adaptive query processing [11, 34, 35]. Our work takes a step towards supporting cost-based query re-optimization in an incre-

mental way, with an ultimate goal of supporting continuous adaptivity in a distributed (e.g., cloud) setting where correlations and runtime costs may be unpredictable at each node. Fine-grained adaptivity has previously only been addressed in the query processing literature via heuristics, such as flow rates [10, 95], which looks at local phenomena rather than long-term cost estimates. For joins and other stateful operators, this has been shown to result in state that later incurs significant costs [33]. Full-fledged cost-based re-optimization can avoid these future costs but has only been possible in a coarse-grained (every few seconds) level [53, 54, 85]. Our dissertation looks at full-fledged cost-based query re-optimization in a systematic way.

Provenance. Provenance (also called lineage) has often been studied to help "explain" why a tuple exists [22] or to assign a ranking or score [18, 40]. Lineage was studied in [30] as a means of maintaining the data in the data warehouses. Our absorption provenance model is a compact encoding of the PosBool provenance semiring in [41] (which provides a theoretical provenance framework, but does not consider implementability). We specialized it for maintenance of derived data in recursive settings. Our approach improves over the counting algorithm [45] which does not support recursion. We have demonstrated the benefits of our approaches in this dissertation versus DRed [45] and maintenance based on relative provenance [40] (both of which were developed for non-distributed query settings).

Declarative approaches. Distributed recursive queries have been proposed as a mechanism for managing state in declarative networks [69]. Our work formalizes aspects of soft-state management and significantly improves the ability to maintain recursive views over dynamic networks. Our distributed recursive view maintenance techniques are applicable to other networked environments, particularly programming abstractions for region-based computations in sensor networks [96, 98]. Our use of declarative techniques to specify the optimizer was inspired in part by the Evita Raced [29] system. However, their work aims to construct an entire optimizer using reprogrammable Datalog rules, whereas our goal is on effective incremental maintenance of the output query plan. We seek to fully match the pruning techniques of conventional optimizers and aim at incremental re-optimization instead of static query optimization.

Incremental view maintenance. Related approaches to incremental maintenance of recursive views and aggregations include graph structured view maintenance have been studied in [102], incremental view maintenance for semi-structured data [4] and for non-distributive aggregate functions [81]. Among these, the closest to our approach is the DRed algorithm [45], which is widely used in conventional centralized framework of handling recursive views. However, it falls short in a distributed framework as its over-deleted and re-derived tuples are too expensive to propagate in many distributed scenarios, as demonstrated in this dissertation.

Chapter 8

Conclusions and Future Directions

To overcome the fundamental limitations of existing approaches to data stream management systems, our dissertation research addresses two important problems: 1) enabling incremental maintenance of *transitive closure* type of queries over distributed data streams; 2) enabling *incremental* query re-optimization, both in a declarative fashion and in a procedural fashion. The first work addresses the *scale* and *performance* issues in supporting transitive closure queries over data streams. It makes practical the support for *linear recursion* in (distributed) data stream management systems. The second work focuses on the *performance* issues of frequent query re-optimizations in stream processing.

Our general thesis is that efficient incremental processing and re-optimization of update streams can be achieved by various *incremental view maintenance* techniques if we cast the problems as incremental view maintenance problems over data streams. In particular, we make the following contributions:

- We formulate the problem of computing recursive tasks over dynamic data streams as a classical *incremental view maintenance* problem, which facilitates many generic incremental view maintenance techniques to address the challenges. We develop a novel, compact *absorption provenance* as an annotation attached to each data item, which enables us to directly detect when view tuples are no longer derivable and should be removed, where the views are defined over a stream of insertions or deletions. We also develop several heuristics to ensure that the absorption provenance annotation, maintained in a Binary Decision Diagram (BDD), remains compact un-

der different topologies. We implement all of the above solutions in our ASPEN prototype system, and experimentally validate the performance under several different scenarios. Results show that we have orders of magnitude savings compared to prior approaches in various settings.

- We explore whether full-fledged cost-based *incremental* techniques for query re-optimization can be developed, where an optimizer would only re-explore query plans whose costs were affected by an updated cardinality or cost value; and whether such incremental techniques could be used to facilitate more efficient adaptivity in dynamic scenarios. We propose a rule-based and declarative approach to query re-optimization. We develop a formulation of query re-optimization as an incremental view maintenance problem, for which we develop novel algorithms like incremental aggregate selection, incremental reference counting and incremental pruning. We implement our solutions in our prototype system, ASPEN, with comprehensive studies of performance against alternative approaches over a diverse set of workloads. Results show that we have an order-of-magnitude performance gains versus non-incremental approaches to query re-optimization.

- We show that our approaches to incremental re-optimization from the declarative perspective can be easily incorporated into traditional *procedural-based* query optimizers (e.g., bottom-up optimizers with dynamic programming and top-down optimizers with branch-and-bounding), without changing their architectures. We study how to design and implement full-fledged cost-based procedural incremental query re-optimization frameworks and present both analytical and empirical results.

- As a case study of this dissertation, we have also developed a prototype system ASPEN (in approximately 80k lines of code), which serves as an end-to-end distributed adaptive query processing system to address the challenges of our applications at hand. This system not only implements all of the solutions in this dissertation, but also serves as the backend of a real campus building application (SmartCIS) at Penn [62, 63].

To extend the dissertation work, we have identified a few promising *future directions*,

which include but are limited to: the extensions of our declarative query optimization framework to multi-core and distributed architectures, cost modeling of plan switching for adaptive stream processing, the problem of determining when to adapt in adaptive stream processing scenarios and adaptive stream processing over federated data centers. We illustrate each of them below.

Rethinking query optimization over multi-core and distributed architectures. A traditional query optimizer is performed on a single machine; however, when more and more data sources are correlated and aggregated, and as multi-query optimization becomes essential, how to re-architect a query optimizer over parallel architectures such as multi-core or distributed environment is particularly important and relevant today. One of the natural extensions of our work on declarative query optimization is to exploit parallelism of computation in query optimization: as we express a query optimizer in a set of rules, we can rely on the distributed execution engine to parallelize the computation. There are lots of research on parallelizing Datalog queries and since a query optimizer can be cast as a specific Datalog program, there are numerous opportunities in optimizing the parallelization of this program. The main challenge of parallelizing query optimization lies in the various dependencies of computations within a query optimizer. Recent work on distributed query optimizer [49] adopts a different approach to our declarative work, which proposes heuristics to allocate the entire search space of candidate plans to different machines. Our declarative paradigm of query optimizer will allow us to cast the state of intermediate computations as *data*, and exploit data-driven approaches to partition the work of computation.

Cost modeling of plan switching for adaptive stream processing. One of the biggest challenges in adaptive query processing is the treatment of *plan switching*, which was not a challenge in conventional engines. Unfortunately, few existing work considered the *cost* of switching plans, e.g., the work to be shared across plans, during query re-optimization. Indeed, in many scenarios, the expensive cost of switching is so dominant that migrating to a more cost-efficient plan might not offset the benefit. On the other hand, in an online scenario, one can only *predict* the future based on past observations, hence one needs to estimate how expensive the plan switching might become given past evidence. By

overlooking the cost of plan switching, a query re-optimization solution would become incomplete as it may pick wrong plans to adapt. One line of future directions of this dissertation is to develop a cost model for plan switching and incorporate this into the entire query re-optimization framework. One can decide *whether* to adapt to a plan which might have better performance for the rest of data streams but is expensive to migrate to.

Determining when to adapt in adaptive stream processing. One of the most important decisions to make in adaptive query processing systems for data streams is determining *when* to adapt. The granularity of adaptation will play a huge role in determining the overall performance of query answering over data streams. Open questions lie in deciding the globally optimal points of adaptation in the life time of data streams, as well as online mechanisms to determine optimal points of adaptations when only part of the data streams have been seen and no predications of the future is available.

Adaptive stream processing across federations of data centers. Today in reality a computing platform may sit across multiple geographically distributed data centers. This brings unseen challenges such as huge costs of data transfer, privacy and security issues, partitioning of plans over the federation, load-balancing and re-partitioning, adaptivity and so forth. As the Map-Reduce paradigm becomes increasingly popular these days, we could enhance the capabilities of cloud processing by leveraging the ideas from this dissertation such as incremental processing of data streams, and extend them in nontrivial ways to address the issues of adaptivity, re-partitioning and federated query optimization, etc.

Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christain Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] Daniel J. Abadi, Samuel Madden, and Wolfgang Lindner. Reed: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[5] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, 2010.

[6] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in oracle rdb. *The VLDB Journal*, 5(4):229–237, 1996.

[7] Arvind Arasu. Linear Road benchmark queries expressed in CQL (Continuous Query Language), available from http://infolab.stanford.edu/stream/cql-benchmark.html, 2003.

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[9] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *VLDB*, 2004.

[10] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[11] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *CIDR*, 2005.

[12] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[13] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Commun. ACM*, 46(2):43–48, February 2003.

[14] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):13–24, 2008.

[15] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, 1987.

[16] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.

[17] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[18] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[19] Beate Bollig, Martin Lobbing, and Ingo Wegener. Simulated annealing to improve variable orderings for obdds. In *International Workshop on Logic Synthesis*, 1995.

[20] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *SIGMOD*, 2007.

[21] Randall E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[22] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[23] Michael Carbin. *Learning effective BDD variable orders for BDD-based program analysis*. PhD thesis, Stanford University, 2006.

[24] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[25] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.

[26] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB*, 1994.

[27] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[28] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, 2007.

[29] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, 2008.

[30] Yingwei Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[32] Alan J. Demers, Johannes Gehrke, Rajmohan Rajaraman, Agathoniki Trigoni, and Yong Yao. The Cougar project: a work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.

[33] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[34] Amol Deshpande, Joseph M. Hellerstein, and Vijayshankar Raman. Adaptive query processing: why, how, when, what next. In *SIGMOD*, 2006.

[35] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. *Adaptive Query Processing*, volume 1 (1). Now Publishers, 2007.

[36] Amol Deshpande and Samuel Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.

[37] Rolf Drechsler, Nicole Gockel, and Bernd Becker. Learning heuristics for obdd minimization by evolutionary algorithms. *Parallel Problem Solving from Nature, LNCS 1141*, pages 730–739, 1996.

[38] Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[39] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[40] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[41] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[42] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. Learning to order bdd variables in verification. *Journal of AI Research*, 18:83–116, 2003.

[43] Modeling topology of large networks. Available from `http://www.cc.gatech.edu/projects/gtitm/`.

[44] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *EDBT*, 1996.

[45] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[46] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase: Complex event processing over streams. *CoRR*, abs/cs/0612128, 2006.

[47] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In *SIGMOD*, 1989.

[48] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.

[49] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing query optimization. *PVLDB*, 1(1):188–200, 2008.

[50] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD*, 1997.

[51] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Quering the Internet with PIER. In *VLDB*, 2003.

[52] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[53] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[54] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD*, 2004.

[55] H. V. Jagadish, Rakesh Agrawal, and Linda Ness. A study of transitive closure as a recursion mechanism. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 331–344, New York, NY, USA, 1987. ACM.

[56] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[57] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[58] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. Adaptively reordering joins during query execution. In *ICDE*, 2007.

[59] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential evaluation of continual queries. Technical Report TR 95-17, University of Alberta, June 1995.

[60] Mengmeng Liu, Zachary G. Ives, and Boon Thau Loo. Enabling incremental query re-optimization. Technical Report MS-CIS-11-11, University of Pennsylvania, 2011.

[61] Mengmeng Liu, Zachary G. Ives, and Boon Thau Loo. Enabling incremental query re-optimization. *CoRR*, abs/1409.6288, 2014.

[62] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. Smartcis: integrating digital and physical environments. In *SIGMOD*, 2009.

[63] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. Demonstrated video available from `http://www.cis.upenn.edu/~zives/aspen/`, 2010.

[64] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. Smartcis: integrating digital and physical environments. *SIGMOD Record*, 39(1):48–53, 2010.

[65] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. Technical Report MS-CIS-08-32, University of Pennsylvania, 2008.

[66] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, 2009.

[67] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Maintaining recursive views of regions and connectivity in networks. *IEEE Trans. Knowl. Data Eng.*, 22(8):1126–1141, 2010.

[68] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD*, 1988.

[69] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *SIGMOD*, 2006.

[70] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, 2005.

[71] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[72] Volker Markl, Vijayshankar Raman, Guy Lohman, Hamid Pirahesh, David Simmen, and Miso Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[73] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., 1st edition, 1998.

[74] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[75] Svilen R. Mihaylov, Marie Jacob, Zachary G. Ives, and Sudipto Guha. A substrate for in-network sensor data integration. In *DMSN*, 2008. Technical report version available as University of Pennsylvania MS-CIS-08-25.

[76] Svilen R. Mihaylov, Marie Jacob, Zachary G. Ives, and Sudipto Guha. Dynamic join optimization in multihop wireless sensor networks. *Proc. VLDB Endow.*, 3(1-2):1279–1290, 2010.

[77] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[78] Vivek Narasayya. TPC-D skewed data generator. Available from `ftp.research.microsoft.com/users/viveknar/tpcdskew`, 1999.

[79] Dushyanth Narayanan, Austin Donnelly, Richard Mortier, and Antony Rowstron. Delay aware querying with Seaweed. In *VLDB*, 2006.

[80] Dan Olteanu and Jiewen Huang. Using obdds for efficient query evaluation on probabilistic databases. In *Proc. SUM*, 2008.

[81] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, 2002.

[82] Suchitra Raman and Steven McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM*, 1999.

[83] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[84] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[85] Elke A. Rundensteiner, Luping Ding, Timothy M. Sutherland, Yali Zhu, Bradford Pielech, and Nishant Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, 2004.

[86] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[87] StreamSQL specification. Available from `www.streamsql.org/pages/documentation.html`, 2008.

[88] Streambase systems, inc. Available from `http://www.streambase.com/`.

[89] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, 1991.

[90] Jeffrey V. Sutherland, Willem-Jan van den Heuvel, Tim Ganous, Matthew M. Burton, and Animesh Kumar. *Future of Intelligent and Extelligent Health Environment*, volume 118, pages 278–312. IOS Press, 2005.

[91] Jeffrey Taft. The intelligent power grid. *Innovating for Transformation: The Energy and Utilities Project*, 6:74–76, 2006. Available from `www.utilitiesproject.com`.

[92] Nicholas E. Taylor and Zachary G. Ives. Reliable storage and querying for collaborative data sharing systems. In *ICDE*, 2010.

[93] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive — a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[94] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.

[95] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[96] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[97] John Whaley. Javabdd library. Available from `http://javabdd.sourceforge.net`.

[98] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, 2004.

[99] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[100] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[101] Matei Zaharia, Mosharaf Chodhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud 10*, 2010.

[102] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.