# Physical Plan Instrumentation in Databases: Mechanisms and Applications

**Fotios Psallidas**

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

**COLUMBIA UNIVERSITY**

2019

# ABSTRACT

## Physical Plan Instrumentation in Databases: Mechanisms and Applications

Fotios Psallidas

Database management systems (DBMSs) are designed with the goal set to compile SQL queries to physical plans that, when executed, provide results to the SQL queries. Building on this functionality, an ever-increasing number of application domains (e.g., provenance management, online query optimization, physical database design, interactive data profiling, monitoring, and interactive data visualization) seek to operate on *how* queries are executed by the DBMS for a wide variety of purposes ranging from debugging and data explanation to optimization and monitoring. Unfortunately, DBMSs provide little, if any, support to facilitate the development of this class of important application domains. The effect is such that database application developers and database system architects either rewrite the database internals in ad-hoc ways; work around the SQL interface, if possible, with inevitable performance penalties; or even build new databases from scratch only to express and optimize their domain-specific application logic over how queries are executed.

To address this problem in a principled manner in this dissertation, we introduce a prototype DBMS, namely, SMOKE, that exposes instrumentation mechanisms in the form of a framework to allow external applications to manipulate physical plans. Intuitively, a physical plan is the underlying representation that DBMSs use to encode how a SQL query will be executed, and providing instrumentation mechanisms at this representation level allows applications to express and optimize their logic on how queries are executed.

Having such an instrumentation-enabled DBMS in-place, we then consider how to express and optimize applications that rely their logic on how queries are executed. To best demonstrate the expressive and optimization power of instrumentation-enabled DBMSs, we express and optimize applications across several important domains including provenance management, interactive data visualization, interactive data profiling, physical database design, online query optimization, and query discovery. Expressivity-wise, we show that SMOKE can express known techniques, introduce novel semantics on known techniques, and introduce new techniques across domains. Performance-wise, we show case-by-case that SMOKE is on par with or up-to several orders of magnitudes faster than state-of-the-art imperative and declarative implementations of important applications across domains.

As such, we believe our contributions provide evidence and form the basis towards a class of instrumentation-enabled DBMSs with the goal set to express and optimize applications across important domains with core logic over how queries are executed by DBMSs.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

**Advisors at Columbia.** Words fall short in helping me describe the positive influence that my advisor, Eugene Wu, has had on my academic and personal life. One of the most important presentation skills that Eugene helped me develop, however, is to always use examples. So, his constant help on articulating my ideas, our endless discussions on research and practice, the intellectual freedom he provided me with across projects, the curiosity he sparked in me over research domains and technical problems, his efforts on making me confident about my work and presentations, and his constant suggestions on balancing my work with personal life, are just a few examples of how instrumental he has been on my academic and personal life. Besides Eugene, Luis Gravano has also been an instrumental advisor on my academic and personal life. Introducing me to the research process, helping me appreciate the attention to detail, and providing me with all the options possible for pursuing my goals, are only just a few examples of how Luis has helped me during my PhD years. Finally, Kenneth Ross has also been instrumental on my work. Our many discussions during the database group meetings at Columbia, his genuine questions on my techniques, and his feedback on my dissertation helped strengthen my work in so many directions.

**Thesis committee.** Besides Eugene; Luis; and Kenneth, I would also like to thank Joseph Hellerstein and Divesh Srivastava for participating in my thesis committee, providing valuable feedback, and helping me better shape the final version of this dissertation.

**CUDBG, WuLab, and peers.** I would also like to thank current and former members of both the Columbia Database Group (CUDBG) and the WuLab Group at Columbia: Orestis Polychroniou, John Paparrizos, Pablo Barrio, Eva Sitaridi, Bingyi Cao, Wangda Zhang, Thibault Sellam, Giannis Karamanolakis, Lampros Flokas, Yiru Chen, Yiliang Shi, Haneen Mohammed, and Yinqi (Bill) Sun. They have all been instrumental in my work through their continuous feedback and our various sorts of discussions. Besides members of CUDBG and WuLab, I would also like to thank Yifan Wu, Sanjay Krishnan, Michael Chess, Nigel Schuster, and Christophe Rimann for their feedback on my work and our collaboration.

**Microsoft.** During my internships at Microsoft I had the chance to meet with great people, researchers, and mentors. I would like to specially thank Bolin Ding, Kaushik Chakrabarti, Surajit Chaudhuri, and Vivek Narasayya for our collaboration and their guidance during my internships at Microsoft Research and the DMX group. I would also like to thank Carlo Curino and the rest of the members of the Cloud and Information Services Lab (CISL) and Gray Systems Lab (GSL) at Microsoft for their valuable feedback on my PhD work.

**University of Athens.** My academic journey started from the University of Athens at the Department of Informatics and Telecommunications. I would like to specially thank Alex Delis, Konstantinos Tzakalozos, and Alexandros Ntoulas for introducing me to research domains and problems and advising me both on my academic and personal life.

**Friends in Greece and NY.** Throughout my PhD years, I was lucky enough to be surrounded by remarkable people and friends both in Greece and in New York. I would like to specially thank Michalis Giannopoulos, Dionysia Kalogeropoulou, Charalampos Chatzicharalampous, Yvonni Nikolaou, Vasilis Stolakis, Maria Tzovara, Orestis Polychroniou, John Paparrizos, and Pablo Barrio for always been a call away.

**Family.** This dissertation would not have been possible without the endless and unconditional support of my family members. I would like to devote a few sentences to specially thank my parents, my sister, my wife, and my new newborn daughter. My parents have always been there for me since day one. There are no words to describe how thankful I am to them for making me the person I am today. My sister has been setting the bar higher for me every single day, challenging me daily to be a better me. Finally, my wife and our newborn daughter have been my everyday heroes and the reason why I smile every day.

To my family.

# Chapter 1

# Introduction

Traditional database management systems are composed out of modules (e.g., parser, logical and physical plan optimizers, and compiler) that work in tandem with the goal set to respond to SQL queries posed by external database clients. Building on this functionality, an ever-increasing number of application domains (e.g., provenance management, online and adaptive physical database design, data profiling, data explanation, auditing, debugging, data visualization, self-regulating and self-tuning databases, and query optimization, to name a few) seek to operate on *how* queries are executed by the database to serve their own application logic. Unfortunately, while each domain is important in its own right, database systems provide little, if any, support for expressing and optimizing applications across such domains. This is primarily because database systems are designed with the goal set to execute SQL queries as opposed to providing mechanisms for external applications to operate on how queries are executed. As a result, expressing and optimizing such important application domains remains challenging with current approaches leading to the development of brittle and expensive techniques through unprincipled workarounds.

To this end, in this dissertation, we focus on a) designing the underlying mechanisms and b) using these mechanisms to express and optimize applications whose core logic relies on how queries are executed. To introduce our overall contributions and provide an outline of the rest of this dissertation, in this introduction we start by providing an overview of key application domains and discussing how the lack of mechanisms is hindering extensibility and optimization per domain (Section 1.1). Then, we summarize how databases are actively being extended in support of the application domains of our focus and outline their inherent limitations (Section 1.2). The limitations of current approaches lead us to pose our main research questions (Section 1.3) and provide an overview (Sections 1.4 and 1.5) of how we aim to address them throughout this dissertation. We conclude our introduction with an outline of the rest of this dissertation and our contributions per chapter (Section 1.6).

## 1.1 Motivating Application Domains

To illustrate how databases are currently being extended in support of application domains that operate on how queries are executed, let us first overview a few motivating domains including provenance management, physical database design, and online query optimization.

### Positive Provenance

Positive provenance, or simply provenance, is a fundamental type of information that describes the relationship between individual input and output data items of a computation. Any workflow-based application that relies on logic over the input-output relationships can be expressed in provenance terms. As such, provenance is (or can be) integral across many domains, including debugging [WMS13; KIT10; IST+15; LDY13; CTV05]; data integration [CWW00]; auditing [EU 18]; security [CWH+17; KIT10]; explaining query results [WM13; WMS12; ROS15; DFG17]; cleaning [CIOP14; HKW+15]; and data visualizations [WS97; WPM+17]. In the context of relational queries, such raw input-output connections are generated as part of data processing within operators of a physical plan. Hence, provenance management systems could in principle capture and analyze provenance information by tightly integrating their logic within the query execution logic, as we will see in Chapter 3. Under the absence of mechanisms for integration of third-party logic within physical plan operators, however, state-of-the-art provenance management systems capture and analyze provenance information by either working around the SQL interface or by rewriting database internals in ad-hoc ways. This leads provenance management systems to either incur high provenance capture costs, high provenance analysis costs, or both, as we elaborate further in Chapter 3. In turn, developers resort to manual implementation of data-intensive applications, such as interactive data visualizations, data profiling, and data debugging, that in principle could be expressed declaratively in provenance terms and optimized as such by provenance-enabled database systems.

## Negative Provenance

Provenance is a type of information that we can use to explain why an output item came into being. Equally important, however, is understanding why a data item is not present in the output or, equivalently, why an input has not contributed to any output. This is a type of information known as negative provenance [CWH$^+$17; HCDN08; CJ09] (missing answers [HCDN08] or why-not provenance [CJ09] are alternative names for negative provenance). Applications of negative provenance include network analytics [CWH$^+$17], data debugging [AHS12], causality [MGS11], and integrity repairs [XZAT18; MGS11] among others. In the context of relational queries, as we will see, negative provenance can be captured and analyzed by rewriting physical operators to produce data flows that are not generated during query execution. This is because, in the context of relational queries, input records do not contribute to output records because of plan operators that are filtering them out (e.g., selections, joins, and set differences). For instance, to get negative provenance for a selection operator, we need to get the records that did not satisfy the selection predicate as opposed to getting the records that satisfied the selection for positive provenance. In other terms, whereas positive provenance can be captured and analyzed by inspecting the data flows generated as part of query processing, negative provenance can be captured and analyzed by inspecting the data flows that were not generated by a physical plan. In the absence of mechanisms to generate such negative data flows from within physical plans and integrate external logic to operate on them, provenance management systems support negative provenance by working around the SQL interface. They do so by rewriting queries to equivalent queries with negation [LLG18; MGMS10; KLZ13] and track positive provenance over them on the way we outlined above. Besides the inherent limitations for capturing and analyzing positive provenance by working around the SQL interface, such systems also need to block the actual query execution even further to wait for provenance capture on rewrites with negations to answer which intermediate operator was responsible for stopping an input record from contributing to the output.

## Online Physical Database Design

The performance of query execution heavily depends on the underlying physical database design (i.e., the physical layout of tables and the choice of views and indexes to create). Supporting an ever-changing query workload with a fixed physical database design may result in poor performance. To account for this problem, online physical database techniques detect changes in the query workload and automatically decide on new physical designs to improve the performance on the workload. To do so, such techniques need monitoring capabilities (e.g., to extract runtime statistics such as CPU consumption or memory pressure) and the ability to piggyback computations in physical plans for "execution feedback" (i.e., to gain insights over the underlying data distribution such as cardinalities [SLMK01] and histogram distributions [AC99; BCG01]). Furthermore, monitoring and gaining execution feedback may add significant overhead to the query execution. Hence, online physical database techniques are in need of scheduling mechanisms to balance between piggybacking and overhead. In the absence of such mechanisms, however, the typical way to introduce an online physical database designer within a database involves rewriting a significant portion of the database engine to introduce the components responsible for managing the online physical database design logic. (e.g., profilers [Pro], feedback caches [SLMK01; BCK$^+$11], and query progress estimators [CNR04; KDCN11]).

## Adaptive Physical Database Design

Similarly to online physical database design, adaptive physical database design is also concerned with restructuring the physical layout of the database. In contrast to online techniques, however, adaptive techniques piggyback the physical design construction within the query execution or defer it (entirely or partially) in-between queries. An illustrative class of techniques in this domain is techniques for database cracking [SDL18; SJD13; PPI$^+$14; HIKY12; IKM07a; IKM07b; IMKG11; KM05] which are concerned with the physical reorganization of columns during selection queries over them so that the execution of future selections over them becomes faster. To avoid the time-consuming construction of complete designs (e.g., sorting columns), that online approaches consider, adaptive techniques are concerned with incremental designs (e.g., database cracking techniques sort columns incrementally). To do so, adaptive physical database design techniques require to integrate their logic for the reorganization of the physical database design logic within physical operators. Furthermore, to implement this logic techniques need programmatic access to the underlying storage for reorganization purposes. Under the absence of such mechanisms, however, adaptive physical database design techniques are typically introduced in a database by introducing new physical operators (e.g., MonetDB provides `cracking` and `sideways` operators) that implement the logic of the initial operator (e.g., selection) along with the reorganization logic. Furthermore, the introduction of such operators involves a considerable rewriting of storage managers to account for physical reorganization during query execution. Finally, while adaptive physical database design techniques are essential for numerous user-facing applications, especially interactive ones which are of main focus in this dissertation, the absence of mechanisms for their implementation in a database limits their production, extension, and deployment considerably. In turn, user-facing applications hinder interactivity, and user engagement in data exploration sessions remains subpar.

### Online Query Optimization

Online query optimization is a domain where techniques recognize the fact that a database optimizer may decide on suboptimal plans due to the absence of exact statistics at optimization time and unawareness of conditions that may arise at run time. To address this problem, techniques in this domain collect knowledge about a query (e.g., CPU and memory consumption statistics, better selectivity and cardinality estimates, or even complete data structures such as bloom filters and hash tables) during its execution and, based on this knowledge, make decisions on how to change a physical plan at runtime. For instance, SMOOTH SCAN [BGIA$^+$18] collects statistics during a selection and decides to change selections scans to index scans, and vice versa, at runtime; ADAPTIVE JOINS [SQL18; Ora17] change nested loop joins to hash joins, and vice versa, at runtime; SIDEWAYS [IT08] and LOOKAHEAD [ZPSP17; PDZ$^+$18] information passing techniques collect information from the execution of one operator to pass it over and optimize other operators in a plan, and PROBABILISTIC PREDICATES [LCKC18; LKC18] change selections applied after expensive machine learning operators to probabilistic ones before the machine learning operators. As illustrated by the these examples, techniques in this domain require mechanisms for integration of their logic within operators to extract operator-specific knowledge, for fine-grained control over the runtime of a physical plan to change their control flow, for access and manipulation of internal state of operators, and for specifying and reacting to events at run time. In the absence of such mechanisms, however, each technique has been implemented in an ad-hoc, database-specific way that is also of little use for introducing new or building on top of current query optimization techniques.

## 1.2 Current Approaches and Limitations

As illustrated by the application domains above, the absence of mechanisms leads developers to implement applications whose logic depends on how queries are executed in two ways:

## Rewriting Databases

The first, and most popular, class of approaches rewrite or even write from scratch, a database engine only to support a domain-specific technique. For instance, we could rewrite the physical operators of a database engine and hardcode techniques to perform provenance capture and analysis. The result, however, is only a database engine that supports provenance management that, while important due to the numerous applications that provenance supports, does not provide any mechanisms to support the development and optimization of other techniques that rely their logic on how queries are executed. In turn, we could change over and over again the database engine to introduce such techniques. Besides the complexities of changing a database engine for every new technique that we need to introduce, this also leads to an important problem of reproducibility. As there are many databases available without common mechanisms for operating on how queries are executed, each technique ends up getting introduced in one database without any way of introducing it in another database without having to rewrite the second database as well. For instance, SmoothScan [BGIA+18] is implemented in PostgreSQL, adaptive joins are implemented in two commercial database engines [SQL18; Ora17], sideways information passing [IT08] was originally implemented in Tukwila [IHW04], lookahead information passing is implemented in Quickstep [PDZ+18], and most database cracking techniques [PPI+14; HIKY12; IKM07a; IKM07b] are implemented in MonetDB [BZN05; Mon15a]. Overall, rewriting databases to support domains whose logic relies on how queries are executed leads to an ever increasing landscape of database systems each supporting a subset of possible techniques, without exposing a principled way for extensibility purposes.

## Working Around The SQL Interface

The second class includes approaches that implement their application logic by working around the SQL interface by rewriting a query to one or more other queries. For instance, logical provenance management systems rewrite a relational query to one or more other queries that produce the outputs of the first query as well as provenance information. As another example, to connect this class of approaches to the previous one, consider having data hosted in PostgresSQL and our application needs database cracking functionality. One approach could be to use SQL to ship data from PostgreSQL to MonetDB because MonetDB supports cracking whereas PostgreSQL does not. The major problem with this class is that not every operation over how queries are executed can be expressed in SQL terms. For instance, whereas provenance capture is possible by rewriting queries, performing database cracking during selections or changing nested loop joins to hash joins during execution are not operations expressible in SQL terms. Putting it differently, a SQL query is a specification of *what* to be executed by a database engine and, by the design of SQL, does not expose any information of *how* a query is executed. Hence, rewriting queries at a level that we have no access on how it will be executed is of little use for the application domains of our focus. Besides expressiveness, this class also comes with significant performance penalties. For instance, as we will see in Chapter 3, provenance is by nature a graph that connects inputs with outputs. Capturing provenance information through SQL queries imposes a relational representation of the provenance graph that is expensive to construct. Furthermore, transferring data from a database A to another database B, just because B supports a feature that A does not, also comes with substantial shipping and data fragmentation costs.

## 1.3 Main Research Questions

Regardless of their inherent limitations, the two classes of approaches discussed above highlight two main characteristics. The first class indicates that database systems need to be extended in support of application domains that operate on how queries are executed. The second class indicates that developers are in need of mechanisms to express their logic over how queries are executed; since SQL is an already established programming API across databases it is natural to attempt to work around it no matter its inherent limitations.

In other terms, these characteristics illustrate that databases need to provide extensibility mechanisms to express and optimize applications domains that operate on how queries are executed by a database. This observation leads us to the following two sets of research questions that also provide a classification of technical challenges that we aim to address throughout this dissertation:

- **Mechanisms.** What are the underlying mechanisms that databases need to provide to facilitate the development of techniques whose core logic depends on how queries are executed—a) without having to rewrite the internals of the database and b) avoiding the limitations of working around the SQL interface (**Q1**)? Furthermore, how should we change the underlying database components in support of such mechanisms (**Q2**)?
- **Applications.** Provided a database engine augmented with such mechanisms, what is its overall expressive and optimization power (i.e., can we use it to express well-known techniques across application domains (**Q3**)?, is it possible to introduce novel semantics on known techniques as well as introduce new techniques across domains (**Q4**)?, and what are the overall performance benefits compared to either hand-writing and hand-tuning the application logic by rewriting the database or working around the SQL interface (**Q5**)?). Furthermore, what are best practices for expressing and optimizing techniques with core logic over how queries are executed (**Q6**)?

Having discussed the main research questions of focus in this dissertation, we next provide an overview of how we aim to address these questions throughout this dissertation.

## 1.4 Mechanisms and SMOKE

To address the first set of questions **Q1** and **Q2**, our main idea is to provide mechanisms for *physical plan instrumentation* from within a database. Intuitively, whereas a SQL query is a specification of *what* needs to be executed by a database and does not encode information of how queries are executed, physical plans are the main underlying representations of SQL queries that encode how queries will be executed. Furthermore, instrumentation is a general software concept under which third-party code is allowed to manipulate a program in a given language. In our case, the given language is the language for composing physical plans (i.e., the physical algebra of the database). Hence, by designing mechanisms for instrumenting physical plans, we can provide principled ways for third-party applications to manipulate how queries are executed without having to alter the underlying database and by avoiding the limitations of manipulating queries at the level of SQL.

To realize our mechanisms and address **Q1**, we have built a prototype instrumentation-enabled database engine, namely, SMOKE. SMOKE exposes a physical plan instrumentation framework (Chapter 6) that overall provides mechanisms for expressing and optimizing external applications with logic over how queries are executed—henceforth, *instrumentation applications*. The underlying mechanisms allow instrumentation applications to a) implement and integrate their logic within the query execution logic (e.g., injecting the logic of positive or negative provenance capture, monitoring, or cracking within physical operators), b) schedule the instrumentation logic, in full or partially, after the execution of physical operators in a physical plan (e.g., deferring the provenance capture logic or the online computation of a cardinality estimate) to avoid the overhead of the instrumentation logic on (parts of) the query execution, c) access the underlying storage either to implement their logic, to change the physical database design, or to change the state maintained by operators in a plan, d) act on the control flow of plans by modifying, adding, removing, or replacing physical operators during query execution (e.g., probabilistic predicates can add their probabilistic selections in a plan or change predicates in a selection), and e) specify and subscribe to events (e.g., the memory used by a hash join has a exceeded a threshold) and react to these events in an application-specific way (e.g., replace hash joins with nested loop joins or compress the underlying hash table to decrease the memory pressure).

Besides the instrumentation mechanisms, in Chapter 6 we also discuss how we changed SMOKE in support of instrumentation with respect to how SMOKE operates under normal query execution. This discussion aims to address **Q2**. The main changes involve the underlying producer-consumer compiler that SMOKE uses to compile physical plans to source code and changes in the underlying physical algebra to integrate third-party logic within physical operators. Regarding the former, SMOKE needs to compile *instrumented physical plans* which are different in nature from physical plans. Regarding the latter, we denote places within physical operators that need to be changed to account for the integration of instrumentation logic. Overall, while we need to make changes on the internals of a database, our changes need to happen only once in support of the development of techniques across numerous application domains. This is in contrast to the class of approaches discussed above that rewrite databases for every new technique that needs to be introduced, and overall highlights the extensibility that instrumentation-enabled database systems could provide.

Finally, note that instrumentation is not a concept unique to SMOKE. For instance, PostgreSQL [Pos13], MySQL [Mys18a; Mys18b], MonetDB [Mal18; Mon15b], and Spark [Spa18] already provide instrumentation capabilities. Unfortunately, these instrumentation capabilities are either coarse-grained (i.e., applications need to operate on plans without any mechanisms to assist them) or domain-specific (e.g., for monitoring or query profiling). In contrast, our goal is to provide domain-agnostic instrumentation mechanisms so that applications can express and optimize their arbitrarily complex instrumentation logic.

## 1.5 Applications

To address the second set of questions **Q3-Q6**, we built on top of SMOKE several applications on the intersection of instrumentation with several domains including (positive and negative) provenance management, interactive data visualizations, interactive data profiling, online query optimization, query discovery, and adaptive physical database design.

While our contributions per domain are briefly discussed next, here we note that, across domains, we highlight how known techniques can be expressed in instrumentation terms—hence, addressing **Q3**. Furthermore, we introduce new instrumentation-based techniques as well as novel semantics on known techniques—hence, addressing **Q4**. Moreover, and perhaps more interestingly, our experiments across domains show that the performance of instrumentation-based techniques ranges from being on par with or improve by several orders of magnitude on the performance of hand-tuned implementations or implementations that work around the SQL interface—hence, addressing **Q5**. Finally, throughout our discussion, we present design principles that we followed to express and optimize our instrumentation-based techniques to demonstrate best practices—hence, addressing **Q6**.

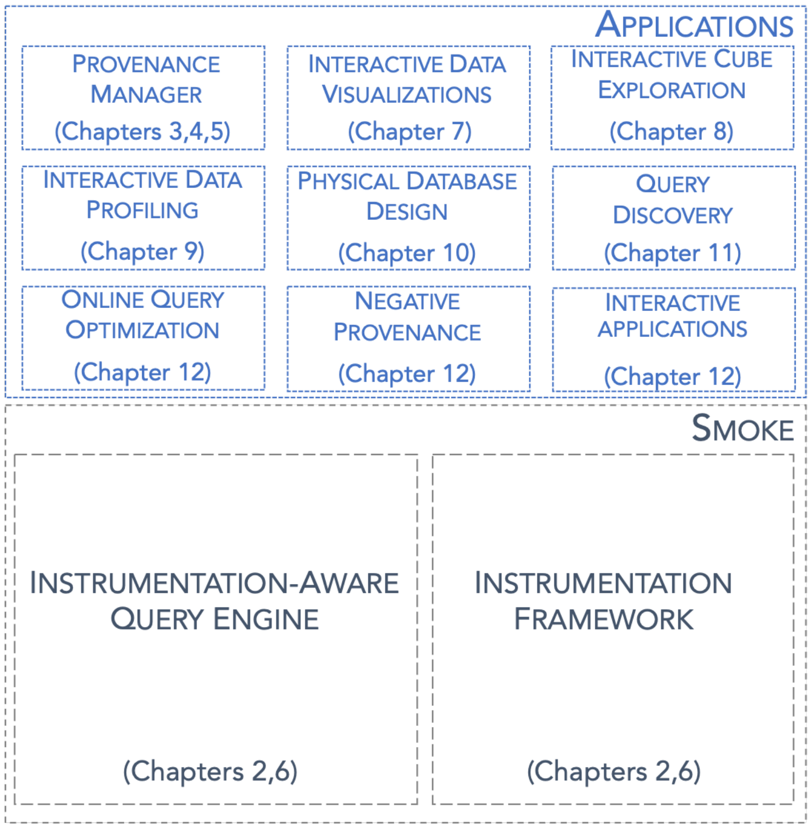

APPLICATIONS
PROVENANCE MANAGER (Chapters 3,4,5)
INTERACTIVE DATA VISUALIZATIONS (Chapter 7)
INTERACTIVE CUBE EXPLORATION (Chapter 8)
INTERACTIVE DATA PROFILING (Chapter 9)
PHYSICAL DATABASE DESIGN (Chapter 10)
QUERY DISCOVERY (Chapter 11)
ONLINE QUERY OPTIMIZATION (Chapter 12)
NEGATIVE PROVENANCE (Chapter 12)
INTERACTIVE APPLICATIONS (Chapter 12)
SMOKE
INSTRUMENTATION-AWARE QUERY ENGINE (Chapters 2,6)
INSTRUMENTATION FRAMEWORK (Chapters 2,6)

Figure 1.1: Dissertation Outline.

## 1.6 Outline and Contributions

To introduce the instrumentation mechanisms of SMOKE and its applications as well as present our overall technical contributions in a meaningful way this dissertation is split into two parts. Figure 1.1 shows the outline of the dissertation graphically. Next, we provide a brief description of the outline of the two parts and summarize our contributions per chapter.

**(Outline of Part I.)** In Part I, we introduce the physical plan instrumentation framework of SMOKE. We do so gradually. First, we introduce the architecture of SMOKE; how it operates under normal query execution and under instrumentation; why our focus is on physical plans as opposed to other representations (e.g., logical plans or source code); and provide some necessary background on physical plans, their compilation to source code in SMOKE, and the history behind SMOKE (Chapter 2). Then, instead of directly presenting the physical plan instrumentation framework, we first present the provenance management components that we have built on top of SMOKE: provenance capture (Chapter 3), evaluation of analytical provenance queries (Chapter 4), and workload-aware optimizations of provenance queries (Chapter 5). Besides our overall contributions on the domain of provenance management, Chapters 3 to 5 serve as an in-depth example of a major instrumentation application (i.e., a provenance manager) that requires several of the components of the physical plan instrumentation framework. Furthermore, as we elaborate in Chapter 2, the first version of SMOKE did not include the instrumentation framework, and the provenance manager was introduced essentially by rewriting the database internals. Hence, having presented our provenance manager in depth and how we introduced it in SMOKE in an ad-hoc way, we introduce the physical plan instrumentation framework in Chapter 6. In doing so, we also highlight its connections with the provenance management techniques of Chapters 3 to 5 and how they could be introduced in a principled way in SMOKE. Overall, Part I aims to address in detail our first set of questions (i.e., **Q1** and **Q2**) and to demonstrate the impact of SMOKE and instrumentation on a major application domain (i.e., provenance management).

**(Outline of Part II.)** Having presented the instrumentation framework and how it can be used in provenance management, in Part II we are concerned with the development of instrumentation applications on top of SMOKE. In Chapter 7, we introduce a language, namely, iSQL, for expressing interactive data visualizations and we draw the connections among provenance (and instrumentation) with interaction classes. To illustrate the performance benefits of SMOKE in the domain of interactive data visualization, in Chapter 8 we introduce instrumentation- and provenance-based techniques for the optimization of crossfiltering [cro15], which is one of the most data-intensive and essential types of interactions. In a similar vein, in Chapter 9, we introduce instrumentation- and provenance-based techniques this time for the optimization of interactive profiling primitives. In Chapter 10, we introduce instrumentation-based techniques and frameworks for deriving both well-known and novel physical database designs. In Chapter 11, we show how we can use an instrumentation-enabled database engine for the discovery of queries of interest based on a novel search interface. Finally, in Chapter 12, we introduce instrumentation-based techniques for expressing query optimization strategies, negative provenance management, and interactive applications as well as propose future work for each technique and domain. Overall, Part II aims to address in detail our second set of questions **Q3-Q5**.

In summary, our contributions and outline per chapter are as follows:

**(Chapter 3.) Fine-Grained Provenance Capture.** In Chapter 3, we introduce a physical algebra that tightly integrates the provenance capture logic within the processing of single and multi-operator plans, and stores provenance in write- and read-efficient indexes. Operators serve the dual purpose of executing the query logic and generating provenance. By doing so, we address the long-standing problem of fine-grained provenance capture to show experimentally orders of magnitude improvements compared to state-of-the-art alternatives. Besides our major contributions on the provenance management domain, with regards to the instrumentation framework, this duality of operators required by provenance capture already highlights the connection between provenance and instrumentation because instrumentation allows us to inject the provenance logic within plans. Whereas in Chapter 3 our discussion does not involve the physical plan instrumentation framework (i.e., provenance capture is introduced by rewriting the internals of the database), our discussion in Chapter 6 sketches how we can induce the same physical algebra using the instrumentation framework.

**(Chapter 4.) Provenance Analytics.** In Chapter 4, we introduce techniques to evaluate analytical provenance queries given the physical representation of provenance as induced by our provenance capture physical algebra in Chapter 3. More specifically, we present techniques for the evaluation of provenance path queries, provenance consuming SQL queries, and provenance semantics (i.e., which-, why-, how- and, where-provenance queries). With regards to instrumentation, this chapter illustrates how instrumentation applications can use instrumentation products (i.e., provenance in this case) to serve their own clients.

**(Chapter 5.) Optimization of Provenance Analytics.** In Chapter 5, we introduce workload-aware optimization techniques for the evaluation of provenance analytics. More specifically, whereas in Chapter 4 we are concerned with the evaluation of provenance queries given the physical representations of provenance from the capture phase of Chapter 3, in Chapter 5 we introduce techniques that optimize the provenance capture phase to induce physical representations targeted to streamlining the evaluation of *known* future provenance queries. With regards to instrumentation, Chapter 5 builds on both Chapters 3 and 4 by showing how we can push the client logic down into the query execution, and highlights that physical operators need to be instrumented in arbitrary ways.

**(Chapter 6.) Physical Plan Instrumentation.** In Chapter 6, we introduce the mechanisms, in the form of a framework, that SMOKE provides for physical plan instrumentation. The underlying mechanisms that the framework exposes can enable applications to implement and integrate their instrumentation logic within the query execution logic; to schedule the instrumentation logic relative to query execution as well as relative to individual operators and pipelines of a physical plan; to modify, add, remove, and replace physical operators at runtime; to specify and react to run time events; and to access and manipulate the underlying storage of SMOKE. Beyond the mechanisms, we also discuss how we changed components of SMOKE (i.e., compiler, physical algebra, and optimizer) in support of instrumentation.

**(Chapter 7.) Expressing Interactive Visualizations.** In Chapter 7, we present iSQL which is our declarative approach towards expressing interactive visualizations. More specifically, iSQL introduces relational data models and query constructs for the specification of interactive visualizations. In this way, iSQL pushes the problem of optimizing interactive data visualizations to the database engine. Unfortunately, even though expressing interactive visualizations in a purely relational manner is sufficient for expressing popular classes of interactive visualizations, optimizing such specifications for several data-intensive classes of interactions (e.g., interactive selections, tooltips and details on demand, linked brushing, and multi-view linking) is hard. To explain why and address this problem, we show how to express these classes in provenance terms. By that, we also show that their relational specification is equivalent to well-known relational specifications of provenance queries that are inherently slow, as we will already have shown in Chapter 3. By expressing these classes in provenance terms, however, allows us to optimize them using the provenance capabilities of SMOKE that avoid the limitations of working around the SQL interface.

**(Chapter 8.) Interactive Cube Exploration and Crossfiltering.** To show experimentally that the provenance mechanisms that SMOKE provides can optimize interactive visualizations, in Chapter 8 we introduce instrumentation- and provenance-based techniques for the optimization of crossfiltering [cro15], which is one of the most data-intensive and essential type of interaction. Our experimental analysis shows that our proposed techniques outperform state-of-the-art approaches on this task all while introducing novel semantics for addressing important problems associated with crossfiltering including the cold-start problem [BCHS17] and the problem of crossfiltering over complicated visualization views.

**(Chapter 9.) Interactive Data Profiling.** In a vein similar to Chapter 8, in Chapter 9 we introduce instrumentation-based techniques this time for the evaluation of data profiling tasks and the interactive exploration of their results. Our techniques cover the evaluation and exploration of functional dependency, uniqueness, and mismatch checks, and our experimental analysis aims to show their performance benefits over state-of-the-art, handwritten implementations for evaluating and exploring the results of data profiling tasks.

**(Chapter 10.) Physical Database Design.** In Chapter 9, we draw the connections between instrumentation and adaptive physical database design. More specifically, we show how instrumentation can assist in the specification of database cracking and adaptive denormalization, which are concerned with performing physical database design during the execution of selections and joins, respectively. Based on these connections, we introduce instrumentation-based frameworks for the introduction of novel cracking techniques within the execution of selection operators, as well as instrumentation-based techniques to adaptively denormalize databases while performing joins. Our experimental analysis over both cracking and denormalization show that instrumentation-enabled engines can express well-known techniques in each of the two domains all while introducing novel semantics and performance benefits towards the robustness of such techniques on future workloads.

**(Chapter 11.) Query discovery.** In Chapter 11, we draw the connections between instrumentation and the space of query discovery from database systems through novel interfaces. More specifically, we introduce a system, namely, S4, that provides a spreadsheet-style keyword search interface that end-users can use to find queries of interests from a database. S4 is a system that precedes SMOKE in its development. As such, besides the algorithmic contributions which are the main focus of Chapter 11, S4 also provides an opportunity for retrospective analysis on how systems could be rethought shall databases like SMOKE expose instrumentation mechanisms. As we will see, most of the time spent in developing S4 and several of its performance benefits come from tasks that could be expressed in SMOKE in a few queries all while meeting the performance provided by S4. This overall highlights the premise of instrumentation-enabled engines in allowing application developers to focus on tasks that are inherent to their goals (e.g., query discovery in this case) as opposed to writing a database from scratch only to embed their logic within physical operators.

**(Chapter 12.) Other connections and the road ahead.** Finally, in Chapter 12, we draw the connections between instrumentation and the domains of online query optimization, negative provenance, and interactive applications. Across domains, we discuss how well-known techniques can be expressed in an instrumentation-based way—hence, further evaluating the expressivity of our instrumentation framework—and we introduce novel extensions and semantics that instrumentation-enabled engines could enable in a principled manner in these domains—hence, covering interesting future directions.

We conclude with a discussion of related work (Chapter 13), implications and takeaways from our current work (Chapter 14), and a summary of future directions (Chapter 15).

# Part I

# Provenance and Instrumentation

# Chapter 2

# Background

In this chapter, we provide necessary background behind our prototype database engine, namely, SMOKE. More specifically, we start by describing the architecture of SMOKE and discuss how SMOKE operates under normal query execution as well as under instrumentation (Section 2.1). Then, we provide necessary background on how SMOKE defines physical plans (Section 2.2) and on the producer-consumer compilation model (Section 2.3). We conclude this chapter with the history behind SMOKE and an outline of the rest of Part I.

## 2.1 Architecture of SMOKE

SMOKE is an in-memory query-compiled database engine augmented with instrumentation capabilities in support of applications ranging from logging to provenance managers and beyond. The main components of SMOKE are illustrated in Figure 2.1.

Next, we first provide an overview of how SMOKE operates under normal query execution. Then, we discuss how SMOKE operates under instrumentation at different intermediate representation (IR) levels (i.e., SQL query, parsed SQL query, logical plan, physical plan, and source code). Finally, we argue on our focus on physical plan instrumentation by discussing limitations on instrumenting at other IR levels for our application domains of focus (i.e., domains whose logic relies on how queries are executed by a database).

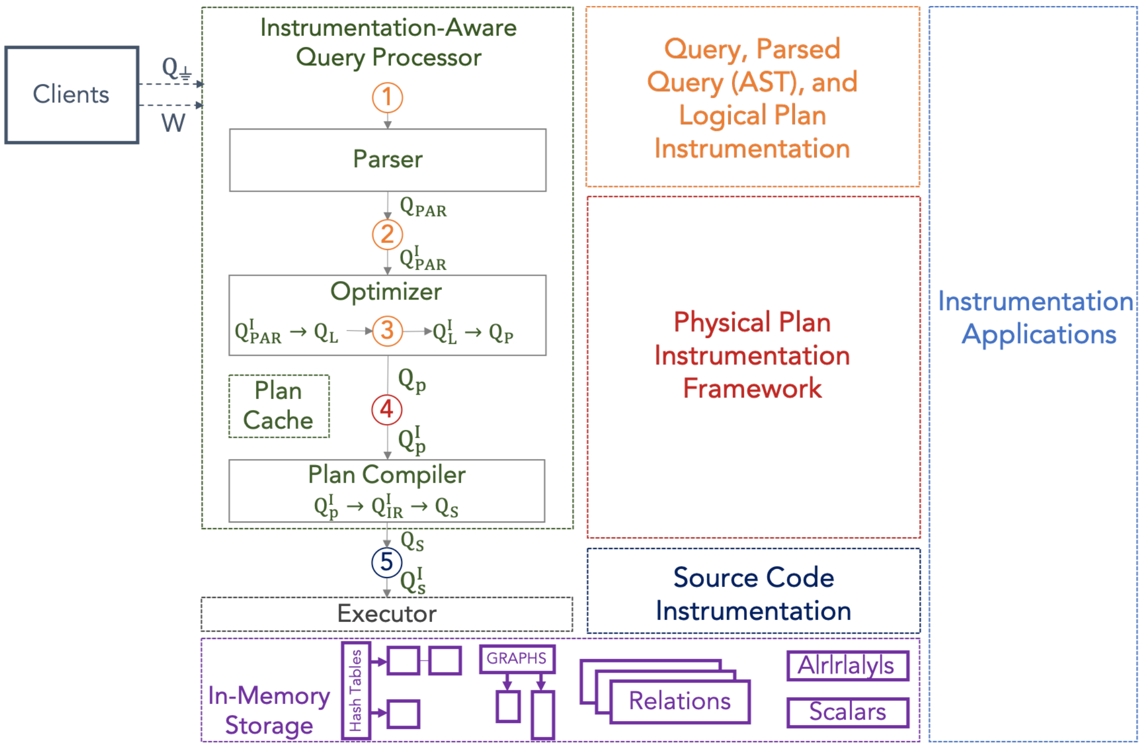

Clients
Q⊥
W
Instrumentation-Aware Query Processor
1
Parser
Q_PAR
2
Q^I_PAR
Optimizer
Q^I_PAR → Q_L → 3 → Q^I_L → Q_P
Plan Cache
Q_P
4
Q^I_P
Plan Compiler
Q^I_P → Q^I_IR → Q_S
Q_S
5
Q^I_S
Executor
Query, Parsed Query (AST), and Logical Plan Instrumentation
Physical Plan Instrumentation Framework
Source Code Instrumentation
Instrumentation Applications
In-Memory Storage
Hash Tables
GRAPHS
Relations
Arrays
Scalars

Figure 2.1: SMOKE architecture: an in-memory query compilation database with instrumentation capabilities on different intermediate representations of SQL queries (i.e., SQL query in textual form, parsed SQL query in abstract syntax tree form, logical plan, physical plan, and source code).

### 2.1.1 SMOKE Under Normal Query Execution

When SMOKE is presented with a query $Q_{\perp}$ from a Client it operates similarly to standard query-compiled databases. First $Q_{\perp}$ will be parsed through the Parser component. The output of the Parser is an AST representation of the $Q_{\perp}$ clauses (e.g., `SELECT`, `FROM`, and `WHERE`). (SMOKE uses and extends the HYRISE parser [FS15] for this step.) The AST is then fed into an in-house, rule-based Optimizer. The Optimizer first converts the AST into a logical plan and then to a physical plan both of which are represented as trees. Logical plans include nodes corresponding to relational operators while physical plans include nodes corresponding to physical operators. The structure of physical plans is important for our discussion throughout this dissertation and we provide background on it in Section 2.2. The overall output of the Optimizer is a physical plan which is fed into the Compiler. The

Compiler follows the producer-consumer compilation model [Neu11] to convert a physical plan to source code. (We also provide background on the compilation model in Section 2.2.) The source code is then sent to the Executor, which simply compiles it down to binary, links the binary with SMOKE, and executes it to compute and send the $Q_{⏚}$ results to the Client.

Throughout this process, a query along with its parsed tree, logical plan, physical plan, source code, and binary representations are stored in the Cache, so that potential future repetitions of it do not have to go through the compilation process again. Finally, note that a Client can also submit a future workload W (i.e., a set of potential future queries; which may be parametrized). In this case, SMOKE will follow the same process for each query in W but it will only store their representations in Cache without executing them.

### 2.1.2 SMOKE Under Instrumentation

Under instrumentation the query processing logic changes. The main difference is that SMOKE introduces points (i.e., ①, ②, ③, ④, and ⑤ in Figure 2.1) where intermediate representations (i.e., queries in textual form, ASTs, logical plans, physical plans, and source code) will be redirected to applications to instrument them. For the purposes of this dissertation, the mechanisms that we provide are focused on instrumenting physical plans through the Physical Plan Instrumentation Framework. However, it is important to provide background on instrumenting at the level of queries, ASTs, logical plans, or source code both to better explain their inherent limitations and because applications that operate on physical plans may be initialized by instrumenting at other IR levels.

Next, we describe how SMOKE operates under instrumentation by explaining the semantics behind the instrumentation points ①, ②, ③, ④, and ⑤ in Figure 2.1. Then, we discuss applications per point and argue on the limitations of instrumenting intermediate representations other than physical plans for our application domains of focus.

① **Query instrumentation.** Whenever a query enters SMOKE, applications that have subscribed to ① will be notified with the SQL specification of the query in textual form, a query id generated by SMOKE (unique for every query), and a timestamp of entrance in the system. The end result of instrumentation at this stage is a query $\mathrm{Q}^{\mathrm{I}}_{⏚}$ that is either the same with $\mathrm{Q}_{⏚}$ or a new query due to rewriting. $\mathrm{Q}^{\mathrm{I}}_{⏚}$ is then fed to the Parser.

2 **AST instrumentation.** Whenever the Parser outputs an AST, denoted as $\mathrm{Q}_{\mathrm{PAR}}$ in Figure 2.1, applications that have subscribed to 2, will be notified with the AST, the query id that corresponds to the query for this AST, and the timestamps of entrance and exit from the Parser. The end result of instrumentation at this stage is an AST $\mathrm{Q}^{\mathrm{I}}_{\mathrm{PAR}}$ that is either the same with $\mathrm{Q}_{\mathrm{PAR}}$ or a new AST due to rewriting.

3 **Logical plan instrumentation.** When the Optimizer takes as input $\mathrm{Q}^{\mathrm{I}}_{\mathrm{PAR}}$, it will first optimize it to generate a logical plan, denoted as $\mathrm{Q}_{\mathrm{L}}$ in Figure 2.1. Applications that have registered to 3, will be notified with the logical plan $\mathrm{Q}_{\mathrm{L}}$, the query id that corresponds to the query for this logical plan, and the timestamps of entrance and exit from the logical plan optimization sub-module of the Optimizer. The end result is a logical plan $\mathrm{Q}^{\mathrm{I}}_{\mathrm{L}}$ that is either the same with $\mathrm{Q}_{\mathrm{L}}$ or a new logical plan due to rewriting.

4 **Physical plan instrumentation.** Given a logical plan $\mathrm{Q}^{\mathrm{I}}_{\mathrm{L}}$, the Optimizer generates a physical plan $\mathrm{Q}_{\mathrm{P}}$. Applications registered to 4, will be notified with the physical plan $\mathrm{Q}_{\mathrm{P}}$, the query id that corresponds to the query for this logical plan, and the timestamps of entrance and exit from the physical plan optimization sub-module of the Optimizer. In contrast to the previous steps, SMOKE provides underlying mechanisms for applications to operate on physical plans, which we cover in Chapter 6. The end result of this step is an instrumented physical plan $\mathrm{Q}^{\mathrm{I}}_{\mathrm{P}}$ which, in contrast to the previous points where applications could only produce the same representation with the ones they were given, is not a physical plan. Intuitively, this is because SMOKE allows instrumented plans to carry the logic of instrumentation applications that is not expressible in the physical algebra of SMOKE.

5 **Source code instrumentation.** Finally, given an instrumented physical plan $\mathrm{Q}^{\mathrm{I}}_{\mathrm{P}}$, the Compiler module will compile it to source code $\mathrm{Q}_{\mathrm{S}}$. Note that the query compilation process for an instrumented physical plan is different from the compilation process for a physical plan, since the instrumented physical plan is not a physical plan per se, as we discussed above. Whether instrumented or not, however, the plan is compiled to source

code and applications registered to ⑤ will be notified with the source code, the query id, and the timestamps of entrance and exit from the Compiler. The end-result of source code instrumentation $\mathrm{Q}_{\mathrm{S}}^{\mathrm{I}}$ is either the same source code as $\mathrm{Q}_{\mathrm{S}}$ (i.e., applications only analyzed $\mathrm{Q}_{\mathrm{S}}$) or another source code due to rewriting.

To conclude our discussion on how SMOKE operates under instrumentation, we finally note that the different IRs (i.e., $\mathrm{Q}_{⏚}$, $\mathrm{Q}_{\mathrm{PAR}}$, $\mathrm{Q}_{\mathrm{L}}$, $\mathrm{Q}_{\mathrm{P}}$, and $\mathrm{Q}_{\mathrm{S}}$) and their instrumented versions (i.e., $\mathrm{Q}_{⏚}^{\mathrm{I}}$, $\mathrm{Q}_{\mathrm{PAR}}^{\mathrm{I}}$, $\mathrm{Q}_{\mathrm{L}}^{\mathrm{I}}$, $\mathrm{Q}_{\mathrm{P}}^{\mathrm{I}}$, and $\mathrm{Q}_{\mathrm{S}}^{\mathrm{I}}$) are stored in the Cache of SMOKE. This is particularly important when instrumenting physical plans because instrumented physical plans can end up generating new operators, as we will see throughout this dissertation. Essentially, instrumenting physical plans allows us to create new physical operators without having to implement and introduce them in the database ourselves, and Cache ends up extending the codebase of the database.

### 2.1.3 Instrumentation at Different IR Levels

Now, as we noted in Chapter 1, our application domains of focus operate on how queries are executed. Not all IRs can support such application domains, however. Having discussed how SMOKE allows instrumentation at different IR levels, we next argue why we focus on physical plan instrumentation by outlining inherent limitations of instrumenting other IRs.

First, SQL queries in textual form, parsed SQL queries in AST form, and logical plans encode *what* needs to be executed by the database. While instrumentation at these IR levels is mainly useful for logging and logical query rewriting, these IRs do not encode *how* queries will be executed by the database. As a result, these IRs provide little information for our application domains of focus (i.e., domains with logic over how queries are executed). For instance, adaptive physical database techniques such as database cracking need to change the physical reorganization at run time. Similarly, query optimization techniques such as adaptive joins need to replace physical operators at run time. These operations cannot be expressed by instrumenting these IRs because they are not expressible in relational terms.

Second, source code encodes information of how queries are executed. Hence, instrumenting at the level of source can express our application domain of focus. The problem, however, is that source code is too low-level (e.g., code blocks with assignment statements) and it is hard to express high-level operations at this level. For instance, applications such as adaptive joins if they need to change a nested loop join to a hash-based join at runtime they need to locate the corresponding loops and variables within the source code, remove them, and replace them with the source code required for hash-join. Implementing such operations is a tedious and error-prone process. This fact highlights that instrumenting at this level does not provide the right level of abstraction for applications to operate on. Furthermore, note that source code instrumentation is only possible in query-compiled engines that generate source code. Source code instrumentation is not possible for interpretation-based engines where the physical plan is what is actually interpreted to perform query execution.

This leaves us with only one option, that of instrumenting physical plans. A physical plan is the first intermediate representation that databases use to encode how a query is executed. Hence, by instrumenting it we can alter, analyze, and create side effects out of how queries are executed which are the main requirements of our application space. Furthermore, in contrast to source code and since we are working on a higher level of abstraction, instrumentation is easier to express and optimize. For instance, to express adaptive joins we can simply replace the nested loops node in the physical plan with an equivalent hash-based one, as we will see in Chapter 6, and let the Compiler take care of the burden of transforming to source code.

So far, we have covered how SMOKE operates under normal query execution and under instrumentation. Next, we provide background on how physical plans are described within SMOKE and how the producer-consumer compilation model operates. (Readers with such background can skip to the end of this section for an overview of next chapters.)

## 2.2 Physical Plans

As we discussed above, we want to provide applications with the ability to instrument physical plans dynamically. This is the main focus of this dissertation. To ease our presentation in subsequent chapters, we now provide background on how physical plans and their underlying physical operators and pipelines are defined within SMOKE.

### Definition of physical plans, operators, and pipelines

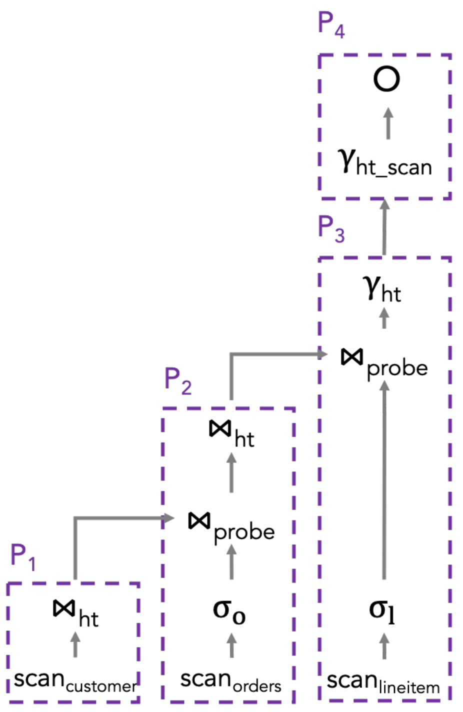
We consider physical plans as trees where each node corresponds to a *physical operator* (drawn from the physical algebra of SMOKE) while directed edges between nodes correspond to data flow through the plan. Furthermore, paths in a physical plan between two blocking operators with all operators in-between to be non-blocking are considered *pipelines*.

```
SELECT   l_orderkey,
         SUM(l_extendedprice*(1-l_discount)) as revenue,
         o_orderdate,
         o_shippriority
FROM     customer, orders, lineitem
WHERE    c_custkey = o_custkey AND
         l_orderkey = o_orderkey AND
         o_orderdate < date '1995-03-05' AND
         l_shipdate > date '1995-03-05'
GROUP BY l_orderkey, o_orderdate, o_shippriority
```

Figure 2.2: TPC-H Q3 variant.

To illustrate these definitions, consider the physical plan in Figure 2.3(a) that is generated by SMOKE for the evaluation of a variant of the TPC-H Q3 query shown in Figure 2.2.

**Example 1 (Physical plan for a variant of TPC-H Q3)** The physical plan is a tree with each node corresponding to a physical operator (e.g., $\gamma_{\text{ht}}$, $\bowtie_{\text{probe}}$, or $\texttt{scan}_{\texttt{lineitem}}$). The plan also contains four pipelines $\mathrm{P}_1, \mathrm{P}_2$, $\mathrm{P}_3$, and $\mathrm{P}_4$ that will be executed in this order. First, $\mathrm{P}_1$ scans the `customer` table ($\texttt{scan}_{\texttt{customer}}$) and generates a hash table on the



(a)

```
void q3(…){
      ht_customer, ht_orders⋈customer, ht_agg
      for t in customer
P1    └ ht_customer.insert(t.custkey,t')

      for o in orders{
        if(!σo(o)) continue;
P2      if(ht_customer.find(o.custkey))
         ht_orders⋈customer.insert(o.orderkey,<o',t'>)
      └}
      for l in lineitem{
        if(!σl(l)) continue;
        if((h = ht_orders⋈customer.find(l.orderkey)))
        {
P3        if((h'=ht_agg.find(gkeys(l,h)))
            insert(ht_agg,l,h.o',h.t')
          else
            update(h',l,h.o',h.t')
        }
      └}

      for h in ht_agg
P4      finalize(h)
      └O.append(make_output(h))
}
```

(b)

Figure 2.3: (a) Physical plan and (b) pseudo-code for the TPC-H Q3 variant.

`customer.custkey` ($\ltimes_{\texttt{ht}}$). Then, $\mathrm{P}_2$ scans the orders table ($\texttt{scan}_{\texttt{orders}}$); filters the `orders` tuples that do not meet the constraint `o_orderdate < date '1995-03-05'` ($\sigma_{\mathrm{o}}$), probes the hash table constructed during $\mathrm{P}_1$ to perform the join on `c_custkey = o_custkey`, and inserts into the hash table ($\ltimes_{\mathrm{ht}}$). $\mathrm{P}_3$ then scans the lineitem table ($\texttt{scan}_{\texttt{lineitem}}$); filters out the tuples that do not meet the constraint `l_shipdate > date '1995-03-05'` ($\sigma_{\mathrm{l}}$); probes the hash table constructed at the end of $\mathrm{P}_2$ to perform the join `l_orderkey = o_orderkey` ($\ltimes_{\texttt{ht}}$); and finally builds a hash table on `l_orderkey, o_orderdate, o_shippriority` that will maintain for each key the value of the sum for the query ($\gamma_{\mathrm{ht}}$). Finally, $\mathrm{P}_4$ scans the hash table to finalize the aggregates ($\mathrm{ht}_{\mathrm{scan}}$) and outputs the result $\mathrm{O}$.

## Description of operators in physical plans

A physical operator can be described by means of describing its inputs and output schema and form, internal state, and whether it is blocking or non-blocking. To be more precise, each physical operator has one or more inputs and one output datasets. (In general, physical operators may have multiple inputs, such as 3-way joins, and multiple outputs, such as partitioning operators. In this dissertation, we will focus only on operators with at most two inputs and one output.) The types of inputs and output datasets can range from scalar values, streams of records, whole tables, views, indexes, hash tables, or other data structures. For instance, the output of $\gamma_{\texttt{ht}}$ in our example is a hash table whereas its input is a stream of records (i.e., the output of $\bowtie_{\mathrm{probe}}$ in $\mathrm{P}_3$). Furthermore, a physical operator can be classified as stateless or stateful depending on whether or not it creates and maintains state during query execution. For instance, $\gamma_{\texttt{ht}}$ is stateful because it maintains and uses a hash table to implement its logic, while $\sigma_{\mathrm{l}}$ in $\mathrm{P}_3$ is stateless because its logic only depends on an input record. Finally, an operator can be blocking (i.e., it needs to consume all of its inputs before producing any output) or non-blocking (i.e., it produces outputs before consuming all of its inputs). For instance, $\gamma_{\texttt{ht}}$ is blocking because it can only produce its output (i.e., the hash table) only after it has consumed its whole input, whereas $\sigma_{\mathrm{l}}$ is non-blocking because it can emit a record that satisfies the selection without needing to consume any more input records.

## Description of pipelines

As we noted above, pipelines are paths in the physical plan tree between two blocking operators with all operators in-between to be non-blocking. Pipelines can be described by a) their order relative to each other and b) the set of individual physical operators that are involved in it. For our TPC-H Q3 example, Figure 2.3 shows that the query has 4 pipelines (i.e., $\mathrm{P}_1, \mathrm{P}_2$, $\mathrm{P}_3$, and $\mathrm{P}_4$) that will be executed in this order, as we discussed in Example 1. Furthermore, each pipeline involves a series of physical operators. For instance, $\mathrm{P}_3$ involves

$\texttt{scan}_{\texttt{lineitem}}$, followed by $\sigma_{\mathrm{l}}$ and $\ltimes_{\mathrm{probe}}$, and concluding with $\gamma_{\texttt{ht}}$. The last operator in a pipeline is traditionally called a *pipeline breaker*.

## Physical plan operator interface

Furthermore, each physical operator needs to implement an interface for query processing purposes that essentially defines how an operator consumes inputs and produces outputs. Popular interfaces include the open-next-close or iterator model (for tuple-at-a time or batch query processing) which is followed by interpretation engines and the consumer-producer model (for push-based query processing) which is followed by query compilation engines. In the rest of the chapter, we fix the interface of focus to the producer-consumer one. It is worth pointing, however, that the instrumentation semantics we introduce are irrespective of the underlying interface, as we discuss in Section 6.10.

```
struct PhysicalOpPNodeDescription{
 StateDescription state;
 bit is_blocking;
 DatasetDescription output;
 DatasetDescription left;
 DatasetDescription right;
};

struct PipelineDescription{
 vec<PhysicalOpDescription> ops;
 DatasetDescription output;
 vec<InputDescription> descr;
};
```

```
struct PhysicalOpPNode{
 PhysicalOpDescription descr;
 PhysicalOpPNode left;
 PhysicalOpPNode right;
 virtual void produce(...);
 virtual void consume(...);
};
struct Pipeline{
 PipelineDescription descr;
 vec<PhysicalOpPNode> ops;
};
struct PhysicalPlan{
 PhysicalOpPNode root;
 vec<Pipeline> pipelines;
};
```

Figure 2.4: (left) Interfaces for the description of physical plan operators (i.e., `PhysicalOpPNodeDescription`) and pipelines (i.e., `PipelineDescription`). (right) Interfaces of physical plan operators (i.e., `PhysicalOpPNode`), pipelines (i.e., `Pipeline`), and physical plans (i.e., `PhysicalPlan`).

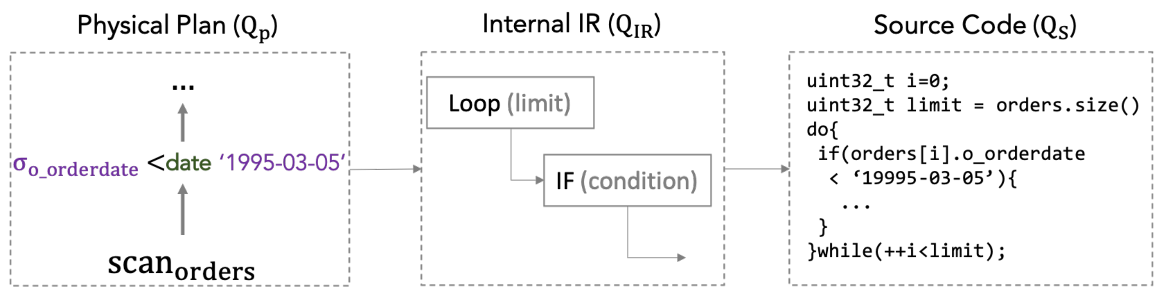
### Implementation in SMOKE

To conclude our discussion on physical plans, Figure 2.4 shows the interfaces that SMOKE uses for describing and implementing plans, pipelines, and physical operators: Figure 2.4(left) shows the interfaces for the description of operators (i.e., `PhysicalOpPNodeDescription`) and pipelines (i.e., `PipelineDescription`), while Figure 2.4(right) shows the interfaces for the definition of physical operators as nodes in physical plans (i.e., `PhysicalOpPNode`), pipelines (i.e., `Pipeline`), and physical plans (i.e., `PhysicalPlan`). Furthermore, the physical algebra of SMOKE includes physical operators in support of materialization, projection, selection, hash-based and nested loop joins, hash-based group-by aggregations, set and bag unions, set and bag intersections, set difference, and cross product. Each individual physical operator is a subclass of `PhysicalOpPNode`; `produce` and `consume` functions implementing the logic of the operator. Next, we provide background on how the Compiler of SMOKE compiles such physical plans to source code.

## 2.3 Physical Plan Compilation in SMOKE

The goal of the SMOKE Compiler is to take as input a physical plan, as specified above, and compile it into source code. To do so, SMOKE follows the producer-consumer compilation model [Neu11; NL14]. Next, we provide background on query compilation of non-instrumented physical plans. (Compilation of instrumented physical plans is covered in Section 6.9 after the introduction of our physical plan instrumentation framework.)

**Example of focus.** To ease our discussion, we consider as an example the compilation of the selection $\sigma_{\mathrm{o\_orderdate=date<`1995-03-05'}}$ over the scan of orders in pipeline $\mathrm{P}_2$ of the physical plan for the TPC-H Q3 variant in Figure 2.3(a). For completeness, the result of compiling the whole TPC-H Q3 variant to source code is shown sketched in Figure 2.3(b).



```
uint32_t i=0;
uint32_t limit = orders.size()
do{
 if(orders[i].o_orderdate
  < ‘19995-03-05’){
   ...
 }
}while(++i<limit);
```

Figure 2.5: Compilation stages of a physical plan in SMOKE.

**Compilation Stages.** SMOKE compiles a physical plan ($Q_P$) by first transforming it to our in-house IR ($Q_{IR}$) and then to source code ($Q_S$). These stages are shown for our example in Figure 2.5 and we discuss them in detail next.

## Physical Plan ($Q_P$) $\rightarrow$ Internal IR ($Q_{IR}$)

Similarly to other query compilation engines that are first compiling physical plans to other IRs [NL14; Neu11; DBCK17; MMP17; KLK+18; TER18] (e.g., internal, LLVM [LA04] or LMS [RO10] ones) before executing, SMOKE also compiles a physical plan to its internal IR. The IR of SMOKE is similar to the ones of LLVM and LMS, but we chose to introduce our own IR so that we can potentially compile to different target languages (e.g., python or R). This is part of future work and we omit further discussion here. For our purposes, SMOKE's internal IR is an abstract syntax tree (AST) with nodes for control flow operations (e.g., loops, conditions, and function blocks) or code blocks each containing a list of code statements in the target IR of the source code.

**Producer-consumer compilation model.** Now, to compile a physical plan to our internal IR, SMOKE uses the producer-consumer compilation model [Neu11; NL14]. Under this compilation model, each physical operator implements its logic in a `produce` and a `consume` function, as we discussed in Section 2.2. Intuitively, an operator asks its children to *produce* their results by calling their corresponding `produce` functions and, in turn, its children produce their results and call the `consume` function of the operator so that it can *consume* the produced results. Finally, each `produce` and `consume` function is implemented in SMOKE using our internal IR. The overall result of this process is getting the logic of the physical plan implemented in our internal IR.

```
class Selection : public PhysicalOpPNode{
 CNF cnf;
 Selection(CNF _cnf):cnf(_cnf){}
 void produce(Context& ctx, Compiler& compiler){
  Required required = make_required(cnf);
  ctx.add(required, this);
  left.produce(ctx, compiler);
 }
 void consume(Context& ctx, Compiler& compiler){
  Required required = ctx.get(this);
  Condition cond = compiler.make_cond(cnf); // creates condition
  compiler.begin_if_statement(cond);       // creates IF node
  if(parent) parent.consume(ctx, compiler);
  compiler.exit_if_statement();
 }
};

class Scan : public PhysicalOpPNode{
 Table tbl;
 Scan(Table _tbl):tbl(_tbl);
 void produce(Context& ctx, Compiler& compiler){
  Variable i = 0;
  Condition limit = compiler.make_condition(i < tbl.limit);
  compiler.begin_loop(i, limit, i++);   // creates Loop node
  RecordVariable r = make_record(tbl, i);  // current row
  serve_required(ctx, tbl, r);
  parent.consume(ctx, compiler);
  compiler.end_loop();
 }
};
```

Figure 2.6: Implementation of `Selection` and `Scan` in SMOKE.

To demonstrate the above compilation process, let us consider how SMOKE compiles our selection example in Figure 2.5(left) to its internal IR in Figure 2.5(middle).

**Example 2 (Compilation Example)** Figure 2.6 sketches the implementation of the `produce` and `consume` functions for the `Selection` and `Scan` operators in SMOKE using our internal IR. The parent operator of the `Selection` (i.e., $\bowtie_{\text{probe}}$ in Figure 2.3) calls the `produce` function of the `Selection`. The `produce` function of the `Selection`, first makes some requirements that it will pass to its child operator (i.e., `Scan` operator). More specifically, the function `make_required(cnf)` (in Line 5) is sugar code for a function that takes as input the `CNF` of the `Selection` and identifies what attributes need to be fetched by the child `Scan` operator. The `required` variable stores what needs to be fetched by the `Scan`, and the `Selection` pushes `required` into the `context` variable (i.e., `context.add(required, this)` in Line 6). The `context` variable maintains what has been required by each node in the physical plan. Finally, the `produce` function of the `Selection` asks its child `Scan` operator to produce (i.e., `left.produce` in Line 7). In turn, the `Scan` operator produces by first generating a scan over the underlying table. It does so by initializing a `Loop` expressed in the internal IR of SMOKE (Lines 22-24). Furthermore, it produces a variable for the underlying record (Line 25) and based on this variable it sets what is required by parent operators (i.e., by binding variables to attributes of the record) within the `serve_required` function (Line 26). For our example, the `Selection` needs the `o_orderdate` to perform the selection, and `serve_required` will bind a variable to this attribute within the record. After having set what was required by parents, the `Scan` asks its parent `Selection` to consume. Note at this point we are still in the `Loop` that performs the scan. Hence, the logic of the parent will be placed within the `Loop`. Now, the `consume` function of the `Selection` is executed (Lines 10-14). First, in Line 10 it gets the variable bindings from `required` (i.e., the variable for the `o_orderdate` attribute that was set by the `Scan`). Using this variable, it creates a condition for the selection. For our example, that would be a condition `o_orderdate < '1995-03-05'` and it is expressed in the internal IR of SMOKE as shown in Line 11. Using this condition, the `Selection` creates an `IF` block, as shown in Line 12. Within the `IF`, the `Selection` asks its parent to consume. Hence, records that pass the `Selection` will be consumed by the parent operator's logic within the `IF` block. Finally, the `Selection` closes the `IF` block, as shown in Line 14. Note, however, that we have not still closed the `Loop` of the scan. After the `consume` function of the `Selection` has returned, the compilation goes back to the `produce` function of the `Scan`, and executes Line 28, which finally closes the `Loop`. The end result of this process for our example, is a `Loop` that scans over the records of the `orders` table with an `IF` block selecting only the records with `o_orderdate < '1995-03-05'`, and the overall program is expressed in the internal IR of SMOKE, as shown in Figure 2.5(middle).

So far, we have discussed how SMOKE compiles a physical plan to its internal IR. Next, we briefly discuss how the IR is compiled down to source code.

### Internal IR ($\mathrm{Q_{IR}}$) $\rightarrow$ Source Code ($\mathrm{Q_S}$)

Compilation in the second stage (i.e., $\mathrm{Q_{IR}} \rightarrow \mathrm{Q_S}$) is straightforward. Recall that our IR is an AST with loop, conditional, code, and function blocks. SMOKE takes the AST and compiles it down to source code directly. For example, the `IF` block with condition `o_orderdate < '1995-03-05'` in our IR is compiled to `if(o_orderdate < '1995-03-05')` in source code (for a suitable definition of the less than operator for dates). Finally, note that for the purposes of this dissertation we only consider compilation to C++ that SMOKE supports in full. Other target IRs as well as optimization steps for our IR are subject to future work.

To conclude our discussion on physical plan compilation, Figure 2.3(b) shows the end result of compiling the plan for the whole TPC-H Q3 variant to source code using the produce-consumer compilation model. Each pipeline corresponds to a for loop within which selections are compiled into if statements (as we showed with our example above), hash table builds have been compiled to their equivalent C++-like hash table builds, and so on. After this compilation step, SMOKE will compile the generated code into machine code (e.g., since the source code is C++, the compilation to machine code will happen using g++) and finally execute the plan to return the result of the query to the Client.

## 2.4 History of Our Proposal and Part Outline

In this chapter, we described how SMOKE operates under normal query processing and under instrumentation of different IRs as well as provided background on physical plans and the producer-consumer compilation model. Our overall discussion focused on the current version of SMOKE. However, historically SMOKE has gone through two major versions with regards to instrumentation, that also highlights main arguments of our discussion behind the introduction of the physical plan instrumentation framework.

More specifically, the normal query execution of SMOKE was first extended to support provenance management. We did so by hard-coding the provenance capture and querying within the physical operators of SMOKE. While important, due to the numerous applications of provenance, the end-result was a database that only supports provenance management. Reflecting back to the two approaches that we discussed in the introduction, we essentially rewrote the internals of a database only to support provenance management in an ad-hoc way. While we were adding more provenance-related functionality (e.g., workload-aware optimizations that we discuss in Chapter 5), it became evident that we had to “pollute” the physical operators with external logic, to the point that the actual logic of the physical operators was dwarfed, making the overall development hard to manage. Hence, to account for a more principled approach towards both provenance as well as other domains whose logic relies on how queries are executed, we introduced physical plan instrumentation.

We believe this historical reflection also highlights main arguments behind the introduction of the physical plan instrumentation framework. In this direction, next we start by describing how to capture provenance (Chapter 3), express and evaluate analytical provenance queries (Chapter 4), and optimize the evaluation of analytical provenance queries in a workload-aware manner (Chapter 5) *without* considering the physical plan instrumentation mechanisms of SMOKE. Then, we conclude Part I by introducing the physical plan instrumentation (Chapter 6) and highlighting its connections with provenance as well as providing simple instrumentation examples from other domains whose logic depends on how queries are executed by a database. In Part II, we delve deeper into applications domains to show further connections and evaluate our instrumentation framework and SMOKE in more detail.

# Chapter 3

# Fine-Grained Provenance Capture

## 3.1 Introduction

Fine-grained provenance, or lineage, describes the relationship between individual input and output data items of a computation. For instance, given an erroneous result record of a workflow, it is helpful to retrieve the intermediate or base records to investigate for errors. Similarly, identifying output records that were affected by corrupted input records can help prevent erroneous conclusions. These operations are expressed as lineage queries over the workflow: backward queries return the subset of input records that contributed to a given subset of output records while forward queries return the subset of output records that depend on a given subset of input records.

Any workflow-based application that relies on logic over the input-output relationships can be expressed in lineage terms. As such, lineage is (or can be) integral across many domains, including debugging [WMS13; KIT10; IST$^+$15; LDY13; CTV05]; data integration [CWW00]; auditing [EU 18]; security [CWH$^+$17; KIT10]; explaining query results [WM13; WMS12; ROS15; DFG17]; cleaning [CIOP14; HKW$^+$15];

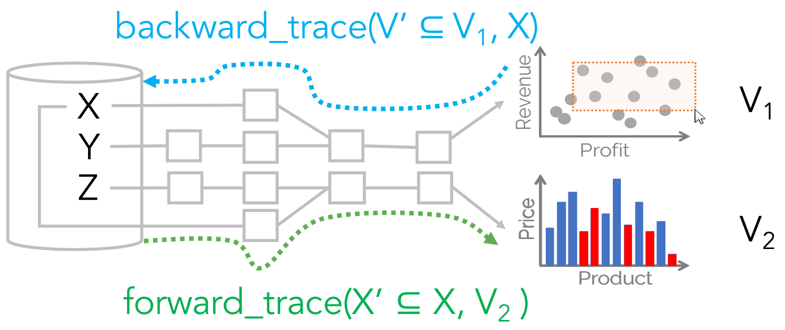


Figure 3.1: Two workflows generate visualizations $V_1$ and $V_2$. A linked brushing interaction highlights in red bars in $V_2$ that share the same input records with selected circles of $V_1$. Logically, it is expressed as a backward query from selected circles in $V_1$ to input tuples followed by a forward query to $V_2$ to highlight bars.

and interactive visualizations as we will see in Chapter 7. This ubiquity highlights the importance of lineage-enabled systems for both traditional as well as emergent domains. To illustrate, consider the lined brushing interactive visualization in Figure 3.1:

**Example 3** Figure 3.1 shows two views $V_1$ and $V_2$ generated from queries over a database. Linked brushing is an interaction technique where users select a set of marks in one view and marks derived from the same records are highlighted in the other views. This functionality is typically implemented imperatively in ad-hoc ways, as we will see in Chapter 7. However, it can be expressed declaratively as lineage queries (i.e., as a backward query from selected circles in $V_1$ to input records, followed by a forward query to highlight corresponding bars in $V_2$) and optimized as such by lineage-enabled systems.

Lineage-enabled systems answer lineage queries by automatically capturing record-level relationships throughout a workflow. A naïve approach materializes pointers between input and output records for each operator during workflow execution and follows these pointers to answer lineage queries. Existing systems primarily differ based on when the relationships are materialized (e.g., *eagerly* during workflow execution or *lazily* reconstructed when executing a lineage query), and how they are represented (e.g., tuple annotations [ABS$^+$06; BCTV04; GA09; IPW11] or explicit pointers [WMS13; LDY13]). Each design trades off between the time and storage overhead to capture lineage, and lineage query performance. For instance, an engine may augment each operator to materialize a hash index that maps output to input records in order to speed up backward lineage queries. However, the index construction costs can dwarf the operator execution cost by $100\times$ or more [WMS13]—particularly if the operator is highly performant.

As data processing becomes faster, a crucial question—and the main focus of this chapter—is whether it is possible to have both negligible lineage capture overhead *and* fast lineage query execution. Unfortunately, current lineage systems incur either high lineage capture overhead, or high lineage query processing costs, or both. Not satisfying these requirements, however, leads developers to abandon declarativity and manually implement lineage-related logic for many data-intensive applications, such as the one in our example.

To this end, we designed the first version of SMOKE (i.e., without any notion of instrumentation) as a fast lineage-enabled in-memory query engine designed to address the major performance overheads in current lineage systems. More specifically, we designed SMOKE based on the careful combination of five design principles that we believe are helpful when incorporating lineage into fast, data-intensive workflow systems. Next, we present the first three of our design principles because they aim to address the problem of fast lineage capture in a workload-agnostic setting (i.e., without knowledge of future queries over lineage) which is the focus of this chapter. The remaining two principles that aim to address the problem of lineage capture in a workload-aware setting are presented in Chapter 5.

**P1. Tight integration.** In high throughput query processing systems, per-tuple overheads incurred within a tight loop—even a single virtual function to store lineage on a separate lineage subsystem [WMS13; IST$^+$15; LDY13]—can slow down operator execution by more than an order of magnitude. In response, SMOKE introduces a new physical algebra that tightly integrates the lineage capture logic into query execution. In addition, SMOKE stores lineage in write-efficient data structures to further reduce the lineage capture overheads.

**P2. Reuse.** Lineage capture introduces significant overhead during query execution due to generating and storing unnecessary amounts of lineage data (e.g., expensive annotations, denormalized forms of lineage). Following the concept of reusing data structures [DBCK17], SMOKE augments and reuses data structures (i.e., hash tables) constructed during normal query execution to overlap capture and execution costs.

**P3. Defer and Inject.** Provenance applications need flexibility with regards to when they could pay the lineage capture costs. For instance, interactive visualization applications may be willing to pay the lineage capture overheads during execution as long as they do not have a negative impact on the interactivity and engagement of end-users. If they have an impact, however, they should be able to defer the lineage capture overheads after query execution (i.e., in-between interactions) given the availability of user think time. In this direction, SMOKE introduces two lineage capture paradigms per physical operator: INJECT and DEFER. The former injects the lineage capture logic within operators by interleaving it with the operator logic; hence the lineage capture overhead is paid during operator execution. The latter defers lineage capture, in full or partially, after the operator execution—hence, paying the lineage capture overhead, in full or partially, after the operator execution.

## Contributions and Chapter Outline

In the rest of the chapter, we start with necessary background (Section 3.2). Then, we present our contributions as follows:

- First, we introduce our write- and read-efficient lineage indexes that SMOKE uses to physically represent fine-grained provenance information. (Section 3.3)
- Then, we introduce a physical algebra that tightly integrates the lineage capture logic within the physical operators that SMOKE supports (i.e., physical operators for the evaluation of the relational operators $\pi$, $\sigma$, $\gamma$, $\bowtie$, $\cup$, $\cap$, $-$, $/$, $\times$, and $\bowtie_{\vartheta}$) and stores lineage in our lineage indexes. For each physical operator we introduce both INJECT

and DEFER semantics. Overall, operators serve the dual purpose of executing the query logic and generating lineage in the form of our lineage indexes. (Section 3.4)

- Furthermore, we extend our support for lineage capture on multi-operator plans by introducing techniques that propagate lineage information throughout plans all while avoiding lineage capture on intermediate physical operators. (Section 3.5)
- Finally, we show experimentally that SMOKE reduces lineage capture overheads *and* lineage query processing costs by up to multiple orders of magnitude compared to state-of-the-art lineage capture and querying approaches. (Sections 3.6 and 3.7)

## 3.2 Problem Definition

Our lineage semantics adhere to the transformation provenance semantics of [CLMR16; GA09; Ike12] over relational queries.

**Base queries.** Formally, let the *base query* $\mathrm{Q}_{\mp}(\mathrm{D}) = \mathrm{O}$ be a relational query over a database of relations $\mathrm{D} = \{\mathrm{R}_1, \cdots, \mathrm{R}_n\}$ that generates an output relation $\mathrm{O}$. An application can initially execute multiple base queries $\mathbb{Q}_{\mp} = \{\mathrm{Q}_{\mp 1}, \cdots, \mathrm{Q}_{\mp m}\}$. For instance, $\mathbb{Q}_{\mp}$ in Figure 3.1 consists of two base queries that generate the two visualization views.

**Lineage queries.** After a base query runs, the user may issue a backward lineage query $\mathrm{L}_b(\mathrm{O}', \mathrm{R}_i)$ that traces from a subset of an output relation $\mathrm{O}' \subseteq \mathrm{O}$ to a base table $\mathrm{R}_i$, or a forward lineage query $\mathrm{L}_f(\mathrm{R}'_i, \mathrm{O})$ that traces from a subset of an input relation $\mathrm{R}' \subseteq \mathrm{R}_i$ to the query's output relation $\mathrm{O}$. The overall result of $\mathrm{L}_b(\bullet)$ and $\mathrm{L}_f(\bullet)$ lineage queries are subsets of input and output relations, respectively.

**Example 4** Let $\mathrm{Q}_{\mp 1}(\{\mathrm{X}, \mathrm{Y}\}) = \mathrm{V}_1$ and $\mathrm{Q}_{\mp 2}(\{\mathrm{X}, \mathrm{Z}\}) = \mathrm{V}_2$ be the base queries in Figure 3.1. The linked brushing interaction is expressed as a backward query $\mathrm{L}_b(\mathrm{V}'_1, \mathrm{X})$ from the selected circles $\mathrm{V}'_1 \subseteq \mathrm{V}_1$ back to the input records in $\mathrm{X}$ that generated them. The forward lineage query $\mathrm{L}_f(\mathrm{L}_b(\mathrm{V}'_1, \mathrm{X}), \mathrm{V}_2)$ retrieves the linked bars in $\mathrm{V}_2$. Since such interactions can expressed in lineage terms, lineage-enabled systems can enable developers to express their application logic declaratively in lineage terms and avoid manual, error-prone implementations. In turn, optimizing lineage constructs from within a lineage-enabled system essentially corresponds to optimizing such interactive data visualization applications.

**Lazy and eager lineage query evaluation.** How can we answer lineage queries quickly? *Lazy* approaches rewrite lineage queries as relational queries over the input relations—the base queries do not incur capture overhead at the cost of potentially slower lineage query processing [Ike12; CWW00; CCT09]. In contrast, we might *Eagerly* materialize data structures during base query execution to speed up future lineage queries [CCT09; Ike12]. We refer to this problem as lineage capture, and we seek to reduce the capture overhead on the base query execution all while speeding up future lineage queries.

**Lineage capture overview.** The eager approach incurs overhead to capture the base query's *lineage graph*. Logically, each edge $\mathrm{a} \overset{\mathrm{op}}{\longleftrightarrow} \mathrm{b}$ maps an operator $\mathrm{op}$'s input record $\mathrm{a}$ to $\mathrm{op}$'s output record $\mathrm{b}$ that is derived from $\mathrm{a}$. Backward lineage connects tuples in the query output $\mathrm{o} \in \mathrm{O}$ with tuples in each input base relation $\mathrm{r} \in \mathrm{R_i}$ by identifying all end-to-end edges $\mathrm{o} \rightsquigarrow \mathrm{r}$ for which a path exists between the two records. Forward lineage reverses these arrows. Materializing such end-to-end forward and backward *lineage indexes* can essentially help us streamline lineage queries (i.e., given subsets of inputs or output we can evaluate lineage queries fast by following edges on the lineage graph).

To address the lineage capture problem, in this chapter we present techniques that efficiently capture lineage by carefully changing implementations of physical operators to both capture lineage as well as provide their regular results. Next, we review alternative techniques that we classify as logical and physical, and we contrast them with our approach.

**Logical lineage capture.** This class of approaches stays within the relational model by rewriting the base query into $\mathrm{Q}'_{\pm}(\{\mathrm{R_1}, \cdots, \mathrm{R_n}\}) = \mathrm{O}'$, so that its output is annotated with additional attributes of input tuples. Some systems [ABS$^{+}$06; CTV05] generate a normalized representation of the lineage graph such that a join query between $\mathrm{O}'$ and each base relation $\mathrm{R_i}$ can create the lineage edges between $\mathrm{O}'$ and $\mathrm{R_i}$. The correct output relation O can be retrieved by projecting away the annotation attributes from $O'$. Alternative approaches [GA09; CTV05] output a single denormalized representation that extends $O'$ with attributes of the input relations. Recent work has shown that the latter rewrite rules (PERM [GA09]) and optimizations leveraging the database optimizer (GPROM [NKG+17]) incurs lower capture overheads than the former normalized approach.

**Physical lineage capture.** This class of approaches instruments physical operators to write lineage edges to a lineage subsystem through an API provided by the subsystem; the subsystem stores and indexes the edges, and answers lineage queries [LDY13; IST+15; WMS13; IPW11; IW10]. This approach can support black-box operators and decouples lineage capture from its physical representation. However, we found that virtual function calls alone (ignoring cross-process overheads) can slow down data-intensive operators by up to $2\times$. Furthermore, lineage capture with external lineage subsystems is not amenable to co-optimization opportunities with the base query execution because the plans generated for lineage capture and query execution are handled by different systems.

**Our approach.** To this end, we designed our lineage capture techniques in SMOKE in ways that avoid the drawbacks of logical and physical approaches. SMOKE improves upon logical approaches by physically representing the lineage edges as read- and write-efficient indexes instead of relationally-encoded annotations. Furthermore, SMOKE improves upon physical approaches by introducing a physical algebra that tightly integrates lineage capture within the logic of physical operators to avoid expensive API calls and in a way amenable to co-optimization with the base query execution due to the tight integration.

Next, we present our techniques by first introducing our read- and write-efficient lineage indexes (Section 3.3), followed by the introduction of our lineage capture techniques on single- and multi-operator physical plans (Sections 3.4 and 3.5, respectively).

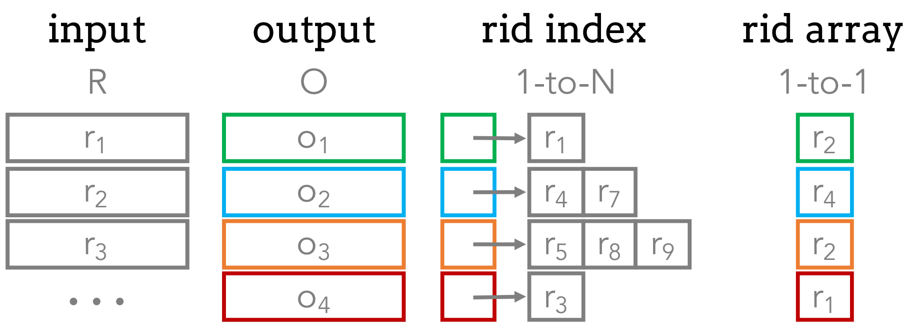


Figure 3.2: Lineage index representations: rid index for 1-to-N (e.g., $\gamma$ backward lineage) and rid array for 1-to-1 (e.g., $\sigma$) relationships.

## 3.3 Lineage Representations

SMOKE uses two main rid-based lineage representations. Figure 3.2 above illustrates input and output relations $R$ and $O$, respectively, and the two rid-based lineage representations for 1-to-N and 1-to-1 relationships between output and input records. We index rids because the indexes are cheap to write (for fast lineage capture) and lookups, that simply index into relations, are fast (for fast lineage query processing). In contrast, indexing full tuples incurs high write costs while indexing primary keys is not beneficial if keys are wide. Furthermore, in-memory engines [ABH+13; FKL+17] already create rid lists, as part of query processing, that resemble our indexes and could be reused for the optimization of lineage capture.

**Rid Index.** 1-to-N relationships are represented as inverted indexes. Consider the backward lineage of `GROUPBY`. The index's $\mathrm{i}^{\mathrm{th}}$ entry corresponds to the $\mathrm{i}^{\mathrm{th}}$ output group, and points to an rid array containing rids of the input records that belong to the group. The rid index can also be used for 1-to-N forward lineage relationships, such as for the `JOIN` operator. Following high-performance libraries [fol17], the index and rid arrays are initialized to 10 elements and grow by a factor of $1.5\times$ on overflow. Our experiments show that array resizing dominates lineage capture costs. Available statistics, however, that allow SMOKE to allocate appropriately sized arrays can reduce lineage capture overheads by up to $60\%$.

**Rid Array.** 1-to-1 relationships between output and input records are represented as a single array. Each entry is an rid rather than a pointer to an rid array as in rid indexes.

## 3.4 Lineage Capture on Single Operator Plans

Having presented the main lineage index representations, in this section we introduce lineage capture techniques to generate lineage indexes when executing individual relational operators. (Section 3.5 extends support to multi-operator plans.) Our techniques are based on two paradigms: INJECT and DEFER (principle **P3** from Section 3.1). DEFER defers portions of the lineage capture until after operator execution while INJECT incurs the full cost during the base query execution. DEFER is preferable when the overhead on the base query execution *must* be minimized or when it is possible to collect cardinality statistics during base query execution to avoid resizing costs. In contrast, INJECT typically incurs lower overall overhead, but the client needs to wait longer to retrieve the base query results.

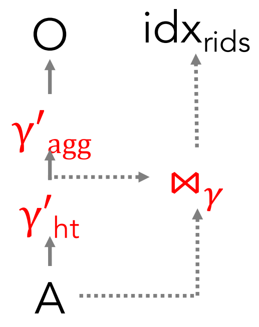
Next, we describe both paradigms for core relational operators. Our discussion also illustrates how both paradigms embody the tight integration and reuse principles (principles **P1** and **P2** from Section 3.1). Our focus is on the mechanisms while Section 3.8 discusses future work to choose between the two paradigms. In our discussion, we introduce DEFER and INJECT provenance capture methods (for all the physical operators supported in SMOKE for the implementation of logical operators including $\pi$, $\sigma$, $\gamma$, $\bowtie$, $\cup$, $\cap$, $-$, $/$, $\times$, $\bowtie_{\vartheta}$) along with code snippets when necessary.

### 3.4.1 Projection

Projection under bag semantics does not need lineage capture because the input and output orders and cardinalities are identical. More specifically, the rid of an output (input) record is its backward (forward) lineage. Projection with set semantics is implemented using grouping and we use the same mechanism as that for group-by aggregation (Section 3.4.3).

### 3.4.2 Selection

Selection is an `if` condition in a `for` loop over the input relation, and emits a record if the predicate evaluates to true [Neu11]. Both forward and backward lineage use rid arrays; the forward rid array can be preallocated based on the cardinality of the input relation. INJECT adds two counters, $\mathrm{ctr_i}$ and $\mathrm{ctr_o}$, to track the rids of the current input and output records, respectively. If a record is emitted, we set the $\mathrm{ctr_i^{th}}$ element of the forward rid array to $\mathrm{ctr_o}$, and append $\mathrm{ctr_i}$ to the backward rid array. Selectivity estimates can be used to preallocate the backward rid array and avoid reallocations during the append operation. DEFER is equivalent to scheduling INJECT after the selection and requires re-scanning the input.



```
Input:  A
Output: O,
        fw[], bw[][]   // forward, backward index
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // γht Build phase
  h = hash(A[i].gbattr)
  if(!ht[h]) ht[h]={init_agg_state(), oid :  -1}
  ht[h].state.update(A[i])

fw = int[A.size()]
bw = int[ht.size()][]
oid = -1;
for h in ht // γagg Scan phase
  O[++oid] = create_output_record(h)
  h.oid = oid

for i=0 to A.size()
  h = hash(A[i].gbattr)
  bw[ht[h].oid].insert(i)
  fw[i] = ht[h].oid
```

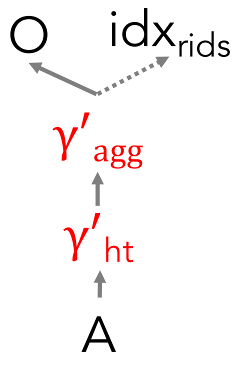
Figure 3.3: Plan (left) and corresponding source code (right) for DEFER fine-grained provenance capture on group-by aggregation.

### 3.4.3 Group-By Aggregation

Query compilers decompose group-by aggregations into two physical operators: $\gamma_{\mathrm{ht}}$ builds the hash table that maps group-by values to their group's intermediate aggregation state; $\gamma_{\mathrm{agg}}$ scans the hash table, finalizes aggregation results for each group, and emits output records. Figures 3.3 and 3.4 show the plans and corresponding source code for the DEFER and INJECT instrumentation paradigms, respectively. Our indexes for group-by aggregation consist of a forward rid array and a backward rid index.

**DEFER.** Consider the DEFER plan and corresponding source code in Figure 3.3. $\gamma'_{\text{ht}}$ for DEFER extends $\gamma_{\text{ht}}$ to store an $\text{oid}$ number to each group's intermediate aggregation state. When $\gamma'_{\text{agg}}$ scans the hash table to construct the output records, it uses a counter to track the output record's rid and assign it to the group's $\text{oid}$ value (i.e., $\text{oid}$ tracks the output rid of the group in the result). SMOKE then pins the hash table in memory. At a later time, $\bowtie_\gamma$ can scan each record in $A$, reuse the hash table to probe and retrieve the associated group's $\text{oid}$, and populate the backward rid index and forward rid array.

Although DEFER must scan $A$ twice, the operator's input and output cardinalities can avoid resizing costs during $\bowtie_\gamma$. Also, $\bowtie_\gamma$ can be freely scheduled (e.g., immediately after $\gamma'_{\text{ht}}$ or during user think time when system resources are free).



```
Input:  A
Output: O,
        fw[], bw[][]   // forward, backward index
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // γht Build phase
  h = hash(A[i].gbattr)
  if(!ht[h]) ht[h]={init_agg_state(), rids=[]}
  ht[h].state.update(A[i])
  ht[h].rids.insert(i)

fw = int[A.size()]
bw = int[ht.size()][]
oid = -1;
for (state, rids) in ht // γagg Scan phase
  O[++oid] = create_output_record(state)
  bw[oid] = rids
  for rid in rids
    fw[rid] = oid
```

Figure 3.4: Plan (left) and corresponding source code (right) for INJECT fine-grained provenance capture on group-by aggregation.

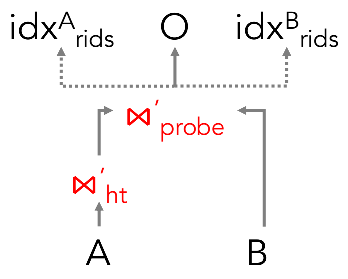
**INJECT.** Consider the INJECT plan and corresponding code snippet in Figure 3.4. $\gamma'_{\text{ht}}$ this time augments each group's intermediate state with an rid array, say, $\text{i}_{\text{rids}}$, which contains the rids of the group's input records (i.e., its backward lineage). $\gamma'_{\mathrm{agg}}$ tracks the current output record id $\mathrm{oid}$ to set the pointer in the backward index to the bucket's rid array and the values in the forward rid array. Since $\gamma'_{\mathrm{agg}}$ knows the input and output cardinalities, it can correctly allocate arrays for the backward and forward indexes. The primary overhead is due to reallocations of $\mathrm{i}_{\mathrm{rids}}$ during the build phase $\gamma'_{\mathrm{ht}}$. As an optimization, our experiments will show that knowing group cardinalities can decrease the capture overhead by up to 60%.

### 3.4.4 Hash-based Joins

SMOKE instruments hash joins in a similar way to hash aggregations. A hash join is split into two physical operators: $\bowtie_{\mathrm{ht}}$ builds the hash table on the left relation $\mathrm{A}$ and $\bowtie_{\mathrm{probe}}$ uses each record of the right relation $\mathrm{B}$ to probe the hash table. Next, we introduce INJECT and DEFER techniques for lineage capture on M:N joins and further optimizations mainly targeting primary key-foreign key joins. For M:N joins, each input record can contribute to multiple output records while each output record is generated from one record of each relation. Hence, SMOKE generates one backward rid array and one forward rid index per input relation.

**INJECT.** Consider the plan and corresponding source code in Figure 3.5 for INJECT provenance capture on joins. The build phase $\bowtie'_{\mathrm{ht}}$ augments each hash table entry with an rid array $\mathrm{i}_{\mathrm{rids}}$ that contains the input rids from $\mathrm{A}$ for that entry's join key. The probe phase $\bowtie'_{\mathrm{probe}}$ tracks the $\mathrm{rid}$ for each output record and populates the forward and backward indexes as expected. Note that output cardinalities are not yet known within the $\bowtie'_{\mathrm{probe}}$ phase and we cannot preallocate our lineage indexes. As a result, although the backward rid arrays are cheap to resize, forward rid indexes can potentially trigger multiple reallocations (i.e., if an input record has many matches) which penalize the capture performance.

**DEFER.** Our main observation is that exact cardinalities needed to preallocate the forward rid indexes are known *after* the probe phase and can be used by DEFER. To this end, DEFER partially defers index construction for the left input relation $\mathrm{A}$ (see Figure 3.6). The build phase adds a second rid array, say, $\mathrm{o}_{\mathrm{rids}}$, to the hash table entry, in addition to $\mathrm{i}_{\mathrm{rids}}$

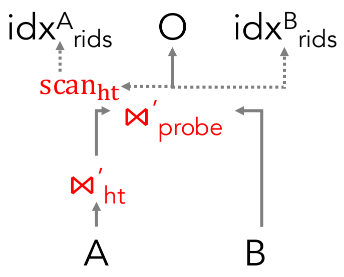


```
Input:  relations A, B;
Output: R                          // A ⋈A.a=B.b B
        a_fw[][], b_fw[][]         // Forward indexes
        a_bw[],   b_bw[]           // Backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()              // Build Phase
  h = hash(A[i].a)
  if(!ht[h]) ht[h]={records=[], i_rids=[]}
  ht[h].records.insert(A[i])
  ht[h].i_rids.insert(i)

o = 0;
for i = 0 to B.size()              // Probe Phase
  h = hash(B[i].b)
  if(!(t = ht.probe(h))) continue;
  for j = 0 to t.i_rids.size()
     R[o] = (t.records[j], B[i])
     a_bw[o] = t.i_rids[j]
     b_bw[o] = i
     a_fw[j].insert(o)
     b_fw[i].insert(o++)
```

Figure 3.5: Plan (left) and corresponding source code (right) for INJECT fine-grained provenance capture on hash-based join.

from INJECT. When $\mathrm{B}$ is scanned during the probe phase, its output records are emitted contiguously, thus $\mathrm{o}_{\mathrm{rids}}$ need only store the rid of the first output record for each match with a $\mathrm{B}$ record. After the $\bowtie'_{\mathrm{probe}}$ phase, the forward and backward indexes for the left relation $\mathrm{A}$ can then be preallocated and populated in a final scan of the hash table (scan$_{\mathrm{ht}}$ in Figure 3.6). Deferring for $\mathrm{B}$ is also possible. However, the benefits are minimal because we need to partition the output records for each hash table entry by the $\mathrm{B}$ records that it matches, which we found to be costly.

**Further optimizations.** If the hash table is constructed on a unique key, then the $\mathrm{i}_{\mathrm{rids}}$ do not need to be arrays and can be replaced with a single integer. Also, if the join is a primary-key foreign-key join, the forward index of the foreign-key table is an rid array. This is because each record of the foreign-key table contributes to exactly one output record. Furthermore, the output cardinality is the same with the foreign-key table cardinality and we preallocate the backward rid array. Finally, join selectivities can help preallocate forward rid indexes,



```
...                             // Build Phase
  if(!ht[h])ht[h]={records=[],i_rids=[],o_rids=[]}
...
o = 0;
for i = 0 to B.size()           // Probe Phase
  h = hash(B[i].b)
  if(!(t = ht.probe(h))) continue;
  t.o_rids.insert(o)
  for j = 0 to t.i_rids.size()
    R[o] = (t.records[j], B[i])
    b_bw[o] = i
    b_fw[i].insert(o++)

a_bw = int[o] // Build indexes for left relation
for h in ht
  s = 0
  for r in h.i_rids
    a_fw[r] = int[h.o_rids.size()])
    for o in h.o_rids
      a_fw[r].insert(o + s)
      a_bw[o+s] = r
    s++
```

Figure 3.6: Plan (left) and corresponding source code (right) for DEFER fine-grained provenance capture on hash-based join.

similarly to how group cardinalities help preallocate backward rid indexes for group-by aggregations.

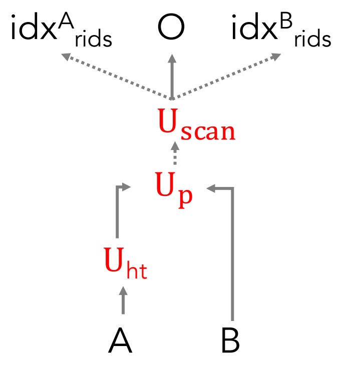
### 3.4.5 Set Union

Set union between two relations $A$ and $B$ (i.e., $A\overset{S}{\bigcup}_{\texttt{uattrs}}B$, where $S$ denotes set union and $\texttt{uattrs}$ denotes the attributes from $A$ and $B$ to union on) are implemented in a hash-based way with consecutive appends to a hash table: Initially, the operator $\cup_{ht}$ builds a hash table using the relation $A$ with the key being the attributes of the union (i.e., $\texttt{uattrs}$). Then, $\cup_{p}$ probes the hash table constructed by $\cup_{ht}$ on the union attributes using relation $B$. If an entry does not already exist for the union attributes, $\cup_{p}$ appends a new entry in the hash table with the union attributes. Essentially, $\cup_{ht}$ and $\cup_{b}$ are the same operator, that probe and append tuples in a hash table. The only difference is that $\cup_{ht}$ takes as input an empty hash table while $\cup_{\mathrm{p}}$ takes as input a pre-built hash table. Finally, $\cup_{\mathrm{scan}}$ scans the hash table and constructs the output.

Regarding provenance on set union, note that each input record (from either $\mathrm{A}$ or $\mathrm{B}$) can contribute to exactly one output record. Furthermore, each output record can be derived by multiple input records due to the semantics of set union. As such, SMOKE generates two backward rid indexes and two forward rid arrays to encode backward and forward provenance. (It is worth emphasizing that just keeping track of the connection between each output and one of the input tuples that contribute to the output does not suffice for provenance purposes. This is primarily because, while all input tuples that contribute to a single output have the same union attributes `uattrs`, the rest of the attributes may be different across these input tuples. Hence, provenance consuming applications may want access to all the input records that contributed to an output to, say, understand their differences.) Next, we discuss DEFER and INJECT approaches for set union; Figures 3.7 and 3.8 show corresponding physical plans and source code for the INJECT and DEFER approaches on set union, respectively, and drive our discussion.

**INJECT.** Figure 3.7 illustrates the INJECT lineage capture of SMOKE for set union. Similarly to group-by aggregation, INJECT rewrites $\cup_{\mathrm{ht}}$ to append, besides the union attributes, two arrays `a_rids` and `b_rids` that track which tuples from $\mathrm{A}$ and $\mathrm{B}$, respectively, contribute to the hash table entry. During $\cup_{\mathrm{ht}}$ we populate `a_rids` and during $\cup_{\mathrm{p}}$ we populate `b_rids`. (SMOKE does so because even though only one copy of unioned tuples is required by the semantics of set union, for provenance purposes we need to keep track of the input tuples that contributed to each output tuple. This information is encoded in the arrays `a_rids` and `b_rids`.) Finally, $\cup_{\mathrm{scan}}$ outputs the result and the provenance indexes.

**DEFER.** Figure 3.8 illustrates the DEFER provenance capture of SMOKE for set union. Similarly to group-by aggregation, DEFER rewrites $\cup_{\mathrm{ht}}$ and $\cup_{\mathrm{p}}$ to append an `oid` to each hash table entry, initially set to $-1$, besides the union attributes. Then, $\cup_{\mathrm{scan}}$ outputs the set union result and assigns the correct oid to each hash table entry. To construct the lineage indexes $\bowtie_{\cup}^{'}$ takes as input the relation $\mathrm{A}$ and probes the previously constructed hash table

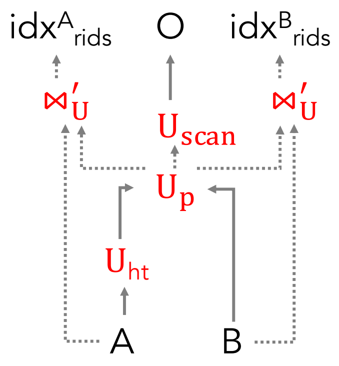


```
Input:  A, B
Output: O,
        a_fw[A.size()], // forward indexes
        b_fw[B.size()]
        a_bw[][], b_bw[][] // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // U_ht: Build phase
  h = hash(A[i].uattrs)
  if(!ht[h]) ht[h]={init_state(A[i].uattrs),
                         a_rids=[], b_rids=[]}
  ht[h].a_rids.insert(i)
for i = 0 to B.size()   // U_p: Probe/Append phase
  h = hash(B[i].uattrs)
  if(!ht[h]) ht[h]={init_state(B[i].uattrs),
                        a_rids=[], b_rids=[]}
  ht[h].b_rids.insert(i)
oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for (state, a_rids, b_rids) in ht //U_scan: Scan phase
  O[++oid] = create_output_record(state)
  a_bw[oid] = a_rids
  for rid in a_rids
    a_fw[rid] = oid
  b_bw[oid] = b_rids
  for rid in b_rids
    b_fw[rid] = oid
```

Figure 3.7: Plan (left) and corresponding source code (right) for INJECT fine-grained provenance capture on set union.

to find the `oid` and properly construct the lineage indexes between the output and input relation $\mathrm{A}$. Similar is the process for $\bowtie'_{\mathrm{U}}$ for the input relation $\mathrm{B}$.

**Further optimizations.** An optimization, for both INJECT and DEFER approaches, is that there is no need to wait to append the right relation $\mathrm{B}$ to the hash table to construct the lineage indexes for the relation $\mathrm{A}$. This is because the intermediate hash table built for $\mathrm{A}$ suffices for the lineage index construction for $\mathrm{A}$. For DEFER, in particular, this also means that the join $\bowtie_{\mathrm{U}}$ for $\mathrm{A}$ will not need to probe a hash table that keeps not all entries for $\mathrm{A}$ but also $\mathrm{B}$. However, this also means that DEFER needs to block the output construction until after the $\bowtie_{\mathrm{U}}$ for $\mathrm{A}$ has been executed, which is a counter-argument to the DEFER paradigm (i.e., lineage is constructed without blocking the query execution). To balance this effect we could keep a copy of the intermediate hash table for $\mathrm{A}$ and use only that for lineage



```
Input:  A, B
Output: O,
        a_fw[A.size()], // forward indexes
        b_fw[B.size()]
        a_bw[][], b_bw[][]  // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // U_ht: Build phase
  h = hash(A[i].uattrs)
  if(!ht[h]) ht[h]={init_state(A[i].uattrs), oid=-1}
for i = 0 to B.size()   // U_p: Probe/Append phase
  h = hash(B[i].uattrs)
  if(!ht[h])  ht[h]={init_state(B[i].uattrs),oid=-1}
oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for h in ht   // U_scan: Scan phase
  O[++oid] = create_output_record(h.state)
  h.oid = oid
for i=0 to A.size() // ⋈′_U: Provenance capture for A
  h = hash(A[i].uattrs)
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
for i=0 to B.size() // ⋈′_U: Provenance capture for B
  h = hash(B[i].uattrs)
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
```

Figure 3.8: Plan (left) and corresponding source code (right) for DEFER fine-grained provenance capture on set union.

construction for the A relation at the cost of copying which could be substantial. SMOKE does not yet support the copy construction, but it does support blocking the set union for the lineage construction.

### 3.4.6 Bag Union

Lineage capture for bag union is simpler than lineage capture for set union. Since for bag union we only concatenate the two input relations, what we only need to maintain is the rid of where one relation ends and the other relation begins in the output of the union. More generally, for bag union of $k$ relations we need $k - 1$ such rids. Using these indexes it is sufficient to answer both backward and forward lineage queries. Note, however, that this lineage capture relies on the fact that the input relation is a base relation stored in the database. For multi-operator plans, the input to the union could be an intermediate relation for which we need to perform lineage capture. For instance, for a query $\sigma_{\vartheta}(\mathrm{A}) \bigcup \mathrm{B}$, we need to perform lineage capture for the selection on $\mathrm{A}$.

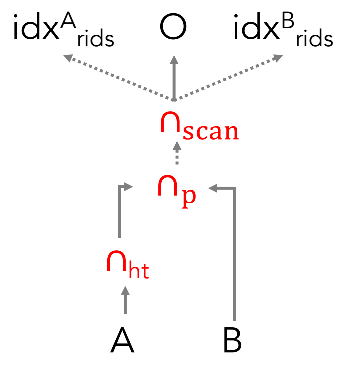
### 3.4.7 Set Intersection

Set intersection in SMOKE is broken into three operators. First, $\cap_{\mathrm{ht}}$ builds a hash table on the outer relation $\mathrm{A}$ with the key being the attributes of the intersection. Each hash table entry, beyond the intersection attributes, also maintains a bit to indicate whether or not it has been matched with a tuple from the inner relation $\mathrm{B}$. Then, $\cap_{\mathrm{p}}$ probes the hash table and sets the bit if a match was found. Finally, $\cap_{\mathrm{scan}}$ scans the hash table and emits the entries to form the output.

Linage capture for set intersection follows the logic of set union. An important difference is that for the INJECT approach, `a_rids` that we have kept for non-matched tuples will be discarded. If the fraction of tuples in the outer relation that appear in the intersection is small that could result in the DEFER approach to be faster than INJECT because it avoids the unnecessary writes in `a_rids`. Also, a slight difference from set intersection without lineage capture, is that INJECT does not require a bit indicating whether a hash table entry has been matched with tuples from the outer relation because we maintain `b_rids` that provide this information. For completeness, Figures 3.9 and 3.10 show physical plans and code snippets for INJECT and DEFER approaches on set intersection, respectively.

### 3.4.8 Bag Intersection

Bag intersection in SMOKE follows the same logic as the set intersection. The only difference is that the hash table needs to maintain two more attributes per entry: (a) the number of tuples from the outer relation that are duplicates according to the intersection attributes, and (b) the number of matches with the inner relation. $\cap_{\mathrm{ht}}$ adds a hash entry {`A[i].iattrs, a_matches=1, b_matches=0`} if there is no prior entry in the hash

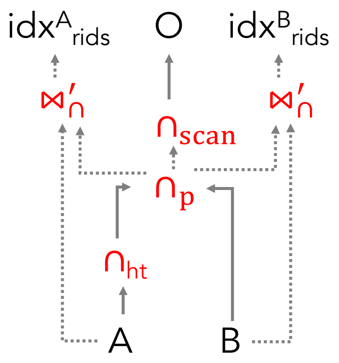


```
Input:  A, B
Output: O,
        a_fw[A.size()], // forward indexes
        b_fw[B.size()]
        a_bw[][], b_bw[][] // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // ∩ht: Build phase
  h = hash(A[i].iattrs)
  if(!ht[h]) ht[h]={init_state(A[i].iattrs),
                           a_rids=[], b_rids=[]}
  ht[h].a_rids.insert(i)

for i = 0 to B.size()   // ∩p: Probe phase
  h = hash(B[i].iattrs)
  if(ht[h]) ht[h].b_rids.insert(i)

oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for (state, a_rids, b_rids) in ht //∩scan: Scan phase
  if(b_rids.size()==0) continue;
  O[++oid] = create_output_record(state)
  a_bw[oid] = a_rids
  for rid in a_rids
    a_fw[rid] = oid
  b_bw[oid] = b_rids
  for rid in b_rids
    b_fw[rid] = oid
```

Figure 3.9: Plan (left) and corresponding source code (right) for INJECT fine-grained provenance capture on set intersection.

table for `A[i].iattrs`, or updates the matches of $A$ (i.e., `a_matches++`) if there was an entry for `A[i].iattrs`. Then, $\cap_p$ probes the hash tables with the tuples from the inner relation $B$ and updates the `b_matches`. Finally, $\cap_{scan}$ scans the hash table and outputs each entry `a_matches·b_matches` times to provide an output with the correct bag semantics.

**INJECT:** Lineage capture for bag intersection under INJECT semantics is straightforward. Instead of keeping `a_matches` and `b_matches` we maintain two arrays of rids (`a_rids` and `b_rids`) from where the matches have originated. As such, `a_matches` = `a_rids.size()` and `b_matches=b_rids.size()`. Hence, $\cap_{scan}$ can still provide an output with the correct bag intersection semantics. Moreover, $\cap_{scan}$ can provide backward

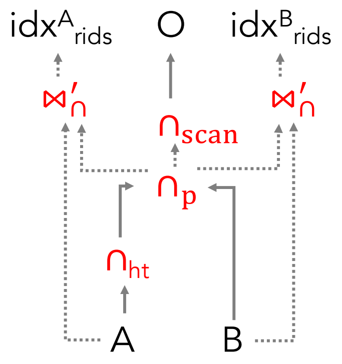


```
Input:   A, B
Output: O,
        a_fw[A.size()], // forward indexes
        b_fw[B.size()]
        a_bw[][], b_bw[][] // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // ∩ht: Build phase
  h = hash(A[i].iattrs)
  if(!ht[h]) ht[h]={init_state(A[i].iattrs),
                          b_bit = 0, oid=-1}

for i = 0 to B.size()   // ∩p: Probe/Append phase
  h = hash(B[i].iattrs)
  if(ht[h]) ht[h].b_bit=1

oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for h in ht   // ∩scan: Scan phase
  O[++oid] = create_output_record(h.state)
  h.oid = oid

for i=0 to A.size() // ⋈'∩: Provenance capture for A
  h = hash(A[i].iattrs)
  if(!h.b_bit) continue
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
for i=0 to B.size() // ⋈'∩: Provenance capture for B
  h = hash(B[i].iattrs)
  if(!h) continue
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
```

Figure 3.10: Plan (left) and corresponding source code (right) for DEFER provenance capture on set intersection.

and forward indexes using these rids. Note, however, that while set intersection has 1-to-N backward lineage, bag intersection has 1-to-1.

**DEFER:** Lineage capture for bag intersection under DEFER follows the logic of DEFER for set intersection. Besides `a_matches` and `b_matches`, each hash entry maintains an output rid `oid` of the first tuple in the output for this hash entry. Note that the output will contain tuples related to this hash entry at rids [`oid`, `oid+a_matches·b_matches`]. Now, the trick is that $\bowtie'_\cap$ need to happen in order first with the A relation and then with B, and for every match we should increase the `oid`. For completeness, Figure 3.11 provides the corresponding plan and code snippet for DEFER provenance capture on bag intersection.



```
Input:  A, B
Output: O,
        a_fw[A.size()], // forward indexes
        b_fw[B.size()]
        a_bw[][], b_bw[][] // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // ∩ht: Build phase
  h = hash(A[i].iattrs)
  if(!ht[h]) ht[h]={init_state(A[i].iattrs),
                    a_matches=1, b_matches=0,
                    oid=-1}
  else ht[h].a_matches++
cnt=0
for i = 0 to B.size()   // ∩p: Probe/Append phase
  h = hash(B[i].iattrs)
  if(ht[h])
    ht[h].b_matches++
    cnt+= a_matches
oid = -1
a_bw = int[cnt][], b_bw = int[cnt][]
for h in ht   // ∩scan: Scan phase
  O[++oid] = create_output_record(h.state)
  h.oid = oid
for i=0 to A.size() // ⋈'∩: Provenance capture for A
  h = hash(A[i].iattrs)
  if(!h.b_matches) continue
  a_bw[ht[h].oid] = i
  a_fw[i] = ht[h].oid++
for i=0 to B.size() // ⋈'∩: Provenance capture for B
  h = hash(B[i].iattrs)
  if(!h) continue
  a_bw[ht[h].oid] =i
  a_fw[i] = ht[h].oid++
```

Figure 3.11: Plan (left) and corresponding source code (right) for DEFER fine-grained provenance capture on bag intersection.

## 3.4.9 Set difference

SMOKE implements set difference of two relations A and B (i.e., $A \overset{S}{\setminus}_{\text{dattrs}} B$) in a hash-based way similar to set intersection. The only differences are (a) we set the `b_bit` of each hash entry to 1 instead of 0 during the initial build and (b) when we probe the hash table with the inner relation we set the `b_bit` to 0 as opposed to 1. The final scan outputs only the hash entries with `b_bit=1` as these are the tuples that appear in the inner relation but do not appear in the outer relation.

Efficient lineage capture for set difference is non-trivial. By definition, the lineage for a tuple $o \in A \setminus B$ depends on (a) the set of tuples in $A$ that it came from and (b) the whole inner relation $B$. Capturing forward indexes for the $A$ tuples follows the lineage capture logic of set intersection and we omit further details. The problem with set difference is that each output depends on the whole outer relation $B$. Our experimental results show that lineage capture is meaningful when lineage has small cardinality. As such, if $B$ is a base relation we do not capture lineage and for backward queries that require access to $B$ we simply scan $B$. Now, if the input relation is an intermediate relation, then SMOKE performs lineage capture during the execution of the operator whose output is the intermediate relation that is the outer relation to the set difference. Hence, for a backward query on the set difference we can access a base relation that is used to construct the intermediate relation through the backward index of the intermediate relation. More interestingly, a forward query from a tuple of a base relation, that is used in the construction of the intermediate relation that is input to the set difference, is the whole output *times* the amount of tuples it contributes to the intermediate relation. This is because each tuple in the intermediate relation contributes to all the tuples in the output of the set difference.

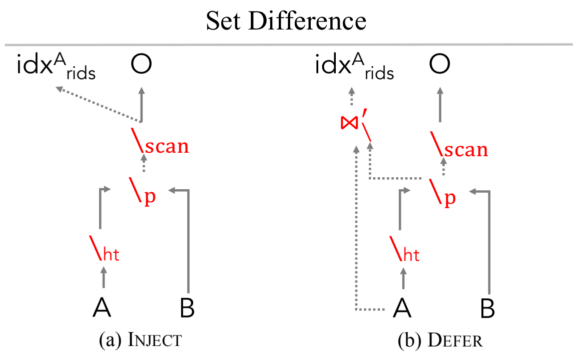
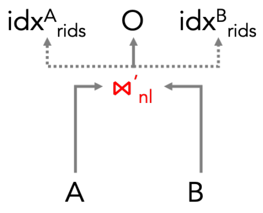


Figure 3.12: INJECT and DEFER plans for set difference.

As such, SMOKE captures lineage only for the A relation that follows the logic of lineage capture for the inner relation of set intersection. For completeness, Figure 3.12 illustrates the corresponding INJECT and DEFER physical plans.

## 3.4.10 $\vartheta$-joins and Nested Loops

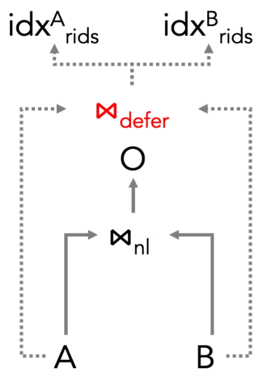
So far, we have proposed a physical algebra for hash-based implementations of equi-joins, group-by aggregations, unions, intersections, and differences. Next, we give a brief discussion for INJECT and DEFER lineage capture on nested-loop based implementations for $\vartheta$-joins. Lineage capture with merge-sort approaches and lineage capture based on nested loops for the rest operators are obvious future work.



```
Input:  A, B
Output: O,
    a_fw[A.size()][], // forward indexes
    b_fw[B.size()][]
    a_bw[], b_bw[]    // backward indexes
oid=-1
for i = 0 to A.size()
  for j = 0 to B.size()
    if(ϑ(A[i], B[j]))
      O[++oid] = create_output_record(A[i], B[j])
      a_bw[oid] = i
      b_bw[oid] = j
      a_fw[i].insert(oid)
      b_fw[j].insert(oid)
```

Figure 3.13: Plan (left) and corresponding source code (right) for INJECT fine-grained provenance capture on nested loop joins.

**INJECT:** Figure 3.13 illustrates the lineage capture of SMOKE for nested-loop $\vartheta$-joins. For each combination of tuples from A and B that satisfy the $\vartheta$ condition the algorithm emits the record to construct the correct output. Since we write serially the output, we can also write serially the lineage indexes and maintain the alignment between each output record and their corresponding backward lineage index. As an optimization, note that the backward index for the A relation can be condensed. All the output records due to A[i] will be consecutive in the output. Hence, instead of keeping the rids for each output `a_fw[i].insert(oid)` we can simply store the rid of only the first one.



```
Input:  A, B
Output: O,
    a_fw[A.size()][], // forward indexes
    b_fw[B.size()][]
    a_bw[], b_bw[]    // backward indexes
oid=-1
for i = 0 to A.size()
  for j = 0 to B.size()
    if(ϑ(A[i], B[j]))
      O[++oid] = create_output_record(A[i], B[j])

oid=-1 // ⋈defer
for i = 0 to A.size()
  for j = 0 to B.size()
    if(<ϑ(A[i], B[j]))><a_bw[++oid] = i><b_bw[oid] =
j><a_fw[i].insert(oid)><b_fw[j].insert(oid)>
```

Figure 3.14: Plan (left) and corresponding source code (right) for DEFER fine-grained provenance capture on nested loop joins.

**DEFER:** Figure 3.14 show the physical plan and corresponding source code for DEFER provenance capture on nested loop joins. The logic for DEFER is similar to the one of INJECT. Instead of materializing our indexes within the nested loop however, we re-execute the nested loop and materialize our indexes during re-execution. Finally, note that during the re-execution we do not propagate results to parents.

### 3.4.11 Cross product

We conclude the introduction of our techniques for fine-grained provenance capture on individual operators by briefly discussing cross products. Regarding cross products, SMOKE does not perform lineage capture in the general case. Given an input tuple from the outer relation $\mathrm{A}$ with rid $\mathrm{a}$ we know that its forward lineage is $\{\mathrm{a}, \mathrm{a}+|\mathrm{B}|, \ldots, \mathrm{a}+(|\mathrm{A}|-1)|\mathrm{B}|\}$ due to the semantics of cross product. Similar is the series for the inner relation. Hence, whether we are given an input or output tuple we can directly infer the backward and lineage rids at runtime without a cost. If the input to cross product is intermediate relations, SMOKE first captures lineage for operators that produce them.

## 3.5 Lineage Capture on Multi-Operator Plans

The naïve way to support multi-operator plans is to individually change each operator to generate its lineage indexes. Lineage queries can then use the indexes to trace backward or forward through the plan. This approach is correct and can support any relational workflow composed out of our physical operators. However, it unnecessarily materializes all intermediate lineage indexes even though only the lineage between output and input records is strictly required for evaluating backward and forward lineage queries.

We address this issue with a technique that a) propagates lineage information throughout plan execution so that only lineage indexes connecting input and output relations are emitted and b) reduces the number of lineage index materialization points in the query plan.

**Lineage propagation.** To propagate lineage throughout plan execution, consider a physical plan with two operators $\mathrm{op_p}$ and $\mathrm{op_c}$ composed as follows: $\mathrm{O} = \mathrm{op_p}(\mathrm{op_c}(\mathrm{R}))$, with input relation $\mathrm{R}$ and output relation $\mathrm{O}$. When $\mathrm{op_p}$ runs, it will use the backward lineage index of $\mathrm{op_c}$ to populate its own lineage index with rids that point to $\mathrm{R}$ rather than the intermediate relation $\mathrm{op_c}(\mathrm{R})$; lineage indexes of $\mathrm{op_c}$ can be garbage collected when not further needed.

**Reduction of materialization points.** To reduce lineage index materialization points, recall that database engines pipeline operators to reduce intermediate results by merging multiple operators into a single pipeline [Neu11]. Operators such as building hash tables are *pipeline breakers* because the input needs to be fully read before the parent operator can run. Within a pipeline, there is no need for lineage capture, but pipeline breakers need to generate lineage along with the intermediate result. In Section 3.4, we showed how pipeline breakers (e.g., hash table construction for the left-side of joins and group-by aggregations) can augment the hash tables with lineage. Parent pipelines that use the same hash-tables for query evaluation (e.g., cascading joins) can also use the lineage indexes embedded in the hash tables to implement the lineage propagation technique above.

**Implementation Details.** Our engine supports naïve lineage capture for arbitrary relational workflows, and we focused our optimizations for SPJA query blocks composed out of pk-fk joins. This was to simplify our engineering and because fast capture for SPJA blocks can be extended to nested blocks by using the propagation technique above. Optimizations for lineage capture across SPJA blocks is interesting future work. We focus on pk-fk joins due to their prevalence in benchmarks and real-world applications and because INJECT and DEFER for pk-fk joins are identical due to our optimizations in Section 3.4.4. Thus, the main distinction between INJECT and DEFER for SPJA blocks is how the final aggregation operator in the block captures lineage; INJECT and DEFER lineage capture on pk-fk joins is identical, while selections and projections are pipelined. Further details are in [PW18].

## 3.6 Experimental Settings

Our experiments in this chapter seek to show that SMOKE (1) incurs significantly lower lineage capture overhead than logical and physical lineage capture approaches and (2) can execute lineage queries faster than lazy, logical, and physical lineage query approaches. To this end, we compare SMOKE to state-of-the-art logical and physical lineage capture and query approaches using microbenchmarks on single operator plans as well as end-to-end evaluations over a subset of TPC-H queries.

**Data.** The microbenchmarks use a synthetic dataset of tables $\texttt{zipf}_{\vartheta,n,g}\texttt{(id,z,v)}$ containing zipfian distributions of varying skew. `z` is an integer that follows a zipfian distribution and `v` is a double that follows a uniform distribution in $[0, 100]$. $\vartheta$ controls the zipfian skew, `n` is the table size, and `g` specifies the number of distinct `z` values (i.e., groups). Tuple sizes are small to emphasize worst-case lineage overheads. End-to-end experiments use the TPC-H data generator and vary the scale factor.

Table 3.1: Lineage capture techniques used in our evaluation.

| **Abbreviation** | **Description** |
|---|---|
| **Smoke** | |
| BASELINE | SMOKE without lineage capture |
| SMOKE-D | SMOKE with defer lineage capture |
| SMOKE-I | SMOKE with inject lineage capture |
| **Logical** | |
| LOGIC-RID | Rid-based annotation |
| LOGIC-TUP | Tuple-based annotation |
| LOGIC-IDX | Indexing input-output relations |
| **Physical** | |
| PHYS-MEM | Virtual emit function calls and no reuse |
| PHYS-BDB | Lineage capture using BerkeleyDB |

To ensure a fair comparison, we implement and optimize alternative, state-of-the-art techniques in our query engine. Our implementation reduces the capture overheads (by several orders of magnitude) as compared to their original implementations, and is detailed in our technical report [PW18].

First, we describe the compared lineage capture techniques (see also Table 3.1 for a brief description of the techniques):

**SMOKE-based techniques. SMOKE-I** and **SMOKE-D** instrument the plan using INJECT and DEFER instrumentation (Section 3.4). Unless otherwise noted, SMOKE-I and SMOKE-D do not use optimizations from Section 3.4. **BASELINE** evaluates base queries on SMOKE without capturing lineage.

**Baseline logical techniques.** State-of-the-art logical approaches (PERM [GA09] and GPROM [NKG+17]) use query rewrites to annotate the base query output with lineage. However, they are built on production databases that incur overheads from transaction and buffer managers, lack of hash-table reuse, and lack of query compilation. These factors could confound results on a system-to-system comparison on the lineage capture problem and would not lead to meaningful results. For this reason, we implemented PERM's rewrite rules (and GPROM's optimizations, whenever applicable) in SMOKE to generate physical plans that annotate output records with either rids (**LOGIC-RID**) or full input tuples (**LOGIC-TUP**). As we noted in Section 3.2, the output annotated relations need to be indexed to support fast lineage lookups. To this end, **LOGIC-IDX** scans the annotated output relation to construct the same end-to-end lineage indexes as those created by SMOKE. For completeness, we also note that our implementation of logical approaches in SMOKE are two orders of magnitude faster than with PERM and GPROM. (Details on how we optimized logical techniques in SMOKE are in [PW18].)

**Baseline physical techniques.** To highlight the importance of tightly integrating lineage capture and operator logic, we use two baseline physical techniques. **PHYS-MEM** instruments each operator to make virtual function calls to store input-output rid pairs in SMOKE lineage indexes from Section 3.3, which highlights the overhead of making a virtual function call for each lineage edge. **PHYS-BDB** instead indexes lineage data in BerkeleyDB to showcase the drawbacks of using a separate storage subsystem [WMS13].

Moreover, we compare lineage querying techniques based on data models and indexes induced during lineage capture:

**Lineage queries.** SMOKE-I, SMOKE-D, LOGIC-IDX, and PHYS-MEM all capture the same lineage indexes from Section 3.3, thus their lineage consuming query performance will be identical. We call this group **SMOKE-L**. We compare with a baseline lazy approach, **LAZY**, which uses standard rules [CWW00; Ike12] to rewrite lineage queries into relational queries that scan the input relations. We also compare with the data model that LOGIC-RID and LOGIC-TUP produce and the indexes that PHYS-BDB generate.

**Measures.** For lineage capture, we report the absolute base query latency and relative overhead compared to not capturing lineage. For lineage queries, we report absolute latency and speedup over baselines. All numbers are averaged over 15 runs, after 3 warm-up runs.

**Platforms.** We ran experiments on a MacBook Pro (macOS Sierra 10.12.3, 8GiB 1600MHz DDR3, 2.9GHz Intel Core i7) and a server-class machine (Ubuntu 14.04, 64GiB 2133MHz DDR4, 3.1GHz Intel Xeon E5-1607 v4). Both architectures have caches sizes 32KiB L1d, 32KiB L1i, and 256KiB L2—the MacBook has 4MiB L3 and the server-class has 10MiB L3. Our overall findings are consistent across the two architectures. Since lineage capture is write-intensive, we report results on the lower memory bandwidth setting (MacBook).

## 3.7 Experimental Results

In this section, we first compare lineage capture techniques on microbenchmarks (Section 3.7.1) and TPC-H queries (Section 3.7.2). Then, we compare techniques on lineage query evaluation (Section 3.7.3).

### 3.7.1 Single Operator Lineage Capture

We first evaluate lineage capture with a set of single operator microbenchmarks for group-by (Section 3.7.1.1), pk-fk joins (Section 3.7.1.2), m:n joins (Section 3.7.1.3), and selections (Section 3.7.1.4). We omit a discussion on other operators to avoid redundant takeaways. Our observations on union, intersection, and set difference are covered by our observations on group-by aggregation, and our observations over cross product and nested loop joins are covered by m:n joins with large fan-out.

#### 3.7.1.1 Group-by Aggregation

We use the base query $\mathrm{Q}_{⏚}$ in Figure 3.16, which groups by $\mathrm{z}$ drawn from a zipfian distribution so that cardinalities are skewed. Semantically, $\mathrm{Q}_{⏚}$ computes multiple statistics following visualization systems that group multiple statistics in a single query [TXS$^+$15]. Figure 3.15 reports the lineage capture latency (base query latency $+$ capture overhead) for each technique while varying the input size (columns) and number of groups (rows).

**Smoke.** SMOKE-I incurs the lowest overhead among techniques ($0.7\times$ on average). SMOKE-D is slightly slower ($1.2\times$ on average) due to the cost of its join $\bowtie_\gamma$ for lineage capture.

**Comparison with logical techniques.** LOGIC-RID and LOGIC-TUP use PERM's aggregation rewrite rule, which computes $\mathrm{Q}_{⏚} \bowtie_\mathrm{z}$ `zipf` to derive the denormalized lineage graph as a single relation. The cost of computing and writing the denormalized lineage graph is costly, slows the base query by multiple orders of magnitude, and is one of the main reasons why SMOKE outperforms alternative logical techniques. Furthermore, since `zipf` is narrow, LOGIC-TUP performs similarly to LOGIC-RID. However, we expect LOGIC-TUP to perform

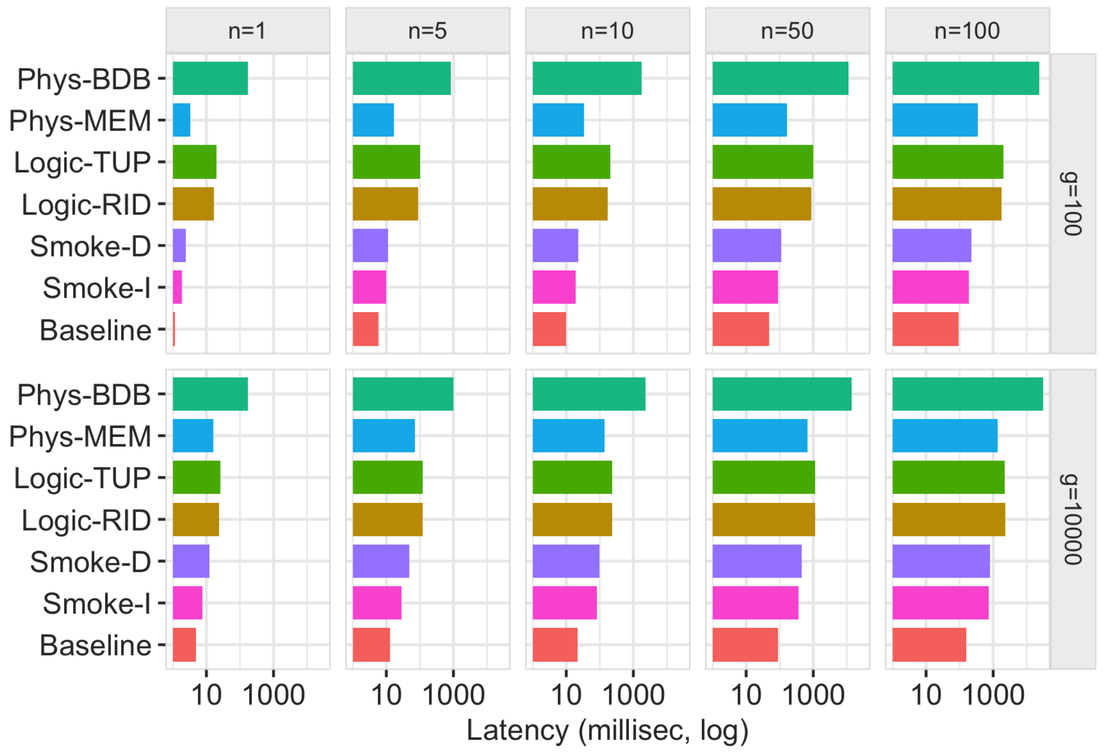


Figure 3.15: Comparison of lineage capture costs for the group-by aggregation operator for different relation cardinalities (columns) and number of distinct groups (rows). SMOKE-I and SMOKE-D slow down the Baseline that does not capture lineage (i.e., it performs only the group-by aggregation) the least as compared to alternative logical and physical capture methods.

```
Q⊥= SELECT    z, COUNT(*), SUM(v), SUM(v*v),
               SUM(sqrt(v)), MIN(v), MAX(v)
    FROM       zipf_{ϑ=1,n,g}
    GROUP BY   z -- #groups follow a zipfian
```

Figure 3.16: Base group-by aggregation query that we use in our lineage capture experiments.

worse for wider input relations. LOGIC-IDX has extra indexing costs over LOGIC-RID and is not plotted.

**Comparison with physical techniques.** The primary overhead for PHYS-MEM is the cost of a virtual function call for each written lineage edge. The cost of building index data structures is comparable to SMOKE's write costs, however SMOKE can reuse the hash table built by $\gamma'_{\mathrm{ht}}$ and incur lower costs for building the backward lineage rid index. PHYS-BDB incurs by far the highest overhead (up to $250\times$ slowdown), due to the overhead of communicating with BerkeleyDB. The same trends hold for the other operators and we have not found physical approaches to be competitive. *As such, we do not report physical approaches in the rest of the experiments.*

**Varying dataset size, skew, and groups.** In general, the lineage capture techniques all incur a constant per input tuple overhead, and differ on the constant value. This is why increasing the input relation size increases costs linearly for all techniques. Increasing the number of groups increases the costs of building and scanning the group-by hash table as well as the output cardinality, and affects all techniques including the baseline. We find that the overhead is independent of the zipfian skew because it does not change the number of lineage edges that need to be written. The skew does affect querying lineage, however, as we will see in Section 3.7.3.

**Complexity of group-by keys and aggregate functions.** We find that the techniques differ in their sensitivity to the size of the group-by keys and the number of aggregation functions in the project clause of the query. SMOKE-I simply generates rid indexes and rid arrays, and is not affected by these characteristics of the base query. In contrast, SMOKE-D and both logical approaches are sensitive to the size of the group-by keys, since they are used to join the output and input relations. Finally, the logical approaches are also affected by the number of aggregation functions because they affect the cost of the final projection. In short, we believe our setup is favorable to alternatives and conclude that SMOKE still shows substantial lineage capture benefits.

**Cardinality Statistics.** SMOKE can also leverage group cardinalities (e.g., through histograms) to allocate correctly sized lineage indexes (Section 3.3). This further reduces the capture overhead by 52% on average and leads to overhead reduction from $0.7\times$ to $0.3\times$ for SMOKE-I (not plotted).

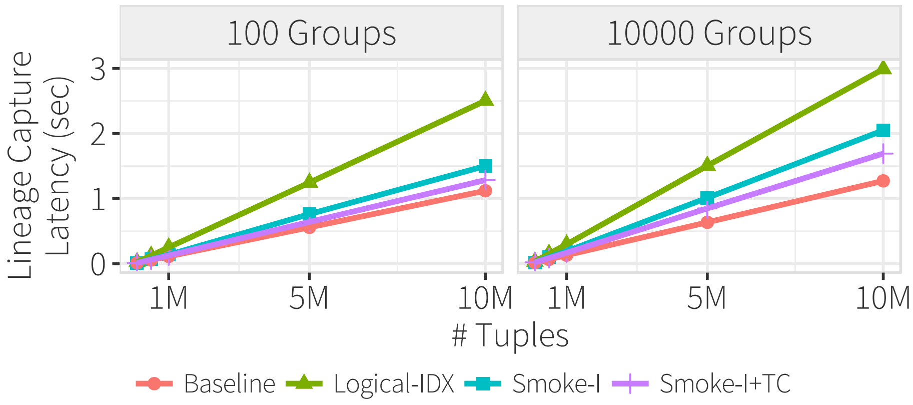


Figure 3.17: SMOKE-I reduces the instrumented pk-fk join latency from 1.4× (LOGIC-IDX) to 0.41×. Knowing the join cardinalities further reduces the overhead to 0.23× (SMOKE-I-TC). SMOKE-D is equivalent to SMOKE-I for pk-fk joins.

$Q_{\Join}$= **SELECT** * **FROM** gids, zipf$_{\vartheta=1,n,g}$
**WHERE** gids.id = zipf$_{\vartheta=1,n,g}$.z

Figure 3.18: Base pk-fk join query that we use in our lineage capture experiments.

#### 3.7.1.2 Primary-Foreign Key (Pk-Fk) Joins

We evaluate lineage capture on pk-fk joins with the base query $Q_{\Join}$ shown in Figure 3.18. `zipf.z` is a foreign key that references `gids.id` and is drawn from a zipfian distribution ($\vartheta = 1$) so that some keys contribute to more join outputs than others. We vary the number of join matches by varying the unique values for `gids.id`. In addition to BASELINE and SMOKE-I, we evaluate SMOKE-I-TC which assumes that we know the number of matches for each `gids.id` and highlights the costs of array resizing. Note that SMOKE-D is equivalent to SMOKE-I due to the pk-fk optimizations in Section 3.4.4. We compare against LOGIC-IDX because LOGIC-RID and LOGIC-TUP do not support forward queries without additional indexes.

**Comparison with logical techniques.** LOGIC-IDX incurs 1.4× capture overhead on average due to the costs of computing and materializing the denormalized lineage graph in

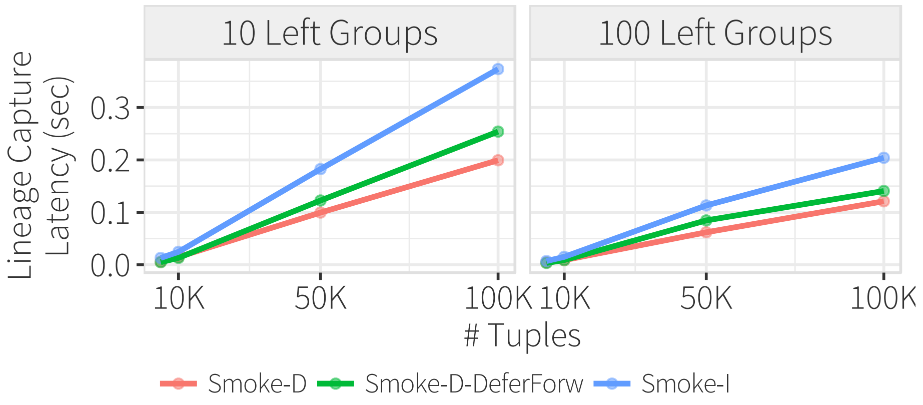


Figure 3.19: M:N join latency when all indexes are populated with SMOKE-I, only forward indexes for the left table are deferred (SMOKE-D-DEFERFORW), and when both lineage indexes are deferred for the left table (SMOKE-D).

```
Q⏚= SELECT *
    FROM zipf1, zipf2
    WHERE zipf1.z = zipf2.z
```

Figure 3.20: Base M:N join query that we use in our lineage capture experiments.

the form of the annotated output relation, and scanning the annotated table to build backward and forward lineage indexes for both input relations. In contrast, SMOKE-I incurs on average $0.41\times$ overhead; knowing join cardinalities reduces the overhead to $0.23\times$ on average. Finally, note that SMOKE-I already knows the cardinalities for the backward indexes and the forward index of the right table for pkfk joins (Section 3.4.4). Thus, the lower overhead of SMOKE-I-TC is due to lower reallocation costs for the forward index of the left table—which is the most expensive index to build due to the 1-to-N relation between primary keys and join outputs.

### 3.7.1.3 Many-to-Many (M:N) Joins

We evaluate lineage capture on M:N joins with the base query $Q_{⏚}$ shown in Figure 3.20. $Q_{⏚}$ here performs a join over the two z attributes drawn from zipfian distributions ($\vartheta = 1$).

`zipf1.z` is within $[1, 10]$ or $[1, 100]$ while `zipf2.z`$\in [1, 100]$. This means that tuples with $\texttt{z} = 1$ have a disproportionately large number of matches compared to larger `z` values that have fewer matches. For this experiment, we also fix the size of the left table `zipf1` to $10^3$ records and vary the right `zipf2` from $10^3$ to $10^5$.

Section 3.4.4 described the INJECT approach for M:N joins, which populates the lineage indexes within the probe phase ($\bowtie_{\text{probe}}$), and the DEFER approach, which computes cardinality statistics during the probe phase to correctly allocate and populate the lineage indexes for the left table after the probe phase to avoid array resizing costs. Finally, to show the benefits of DEFER, we also evaluate SMOKE-D-DEFERFORW which still defers the forward index construction for the left table but populates the backward index within the probe phase. To simplify the presentation, we only report SMOKE-based techniques since our comparisons with alternatives yields findings consistent with the ones presented so far.

**Comparison of SMOKE techniques.** M:N joins over the skewed inputs of our setup are similar to cross-products and yield very large output relations. As a result, the join output materialization dominates the base query execution and renders the lineage capture overheads non-informative. For this reason, here we present results without accounting for the materialization of the output. In this way, the M:N execution is $\approx 0$ms and Figure 3.19 primarily reports lineage capture overhead for the three techniques that we compare. The overheads for SMOKE-I and SMOKE-D-DEFERFORW is predominantly due to resizing. SMOKE-D avoids resizing and reduces the capture overhead the most (up to $2.65\times$). Finally, increasing the number of groups for `zipf1.z` reduces the costs of all techniques because the output cardinality is smaller but the relative capture overheads are the same.

#### 3.7.1.4 Selection

This experiment uses the following base query: `SELECT * FROM zipf WHERE v < ?`, where the attribute `v`$\in [0, 100]$ is drawn from a uniform distribution. Varying the parameter `?` allows us to vary the query selectivity. Figure 3.21 reports the lineage capture costs for two relation sizes ($1, 5$ million), and varying the estimated query selectivity between $1\%$ and

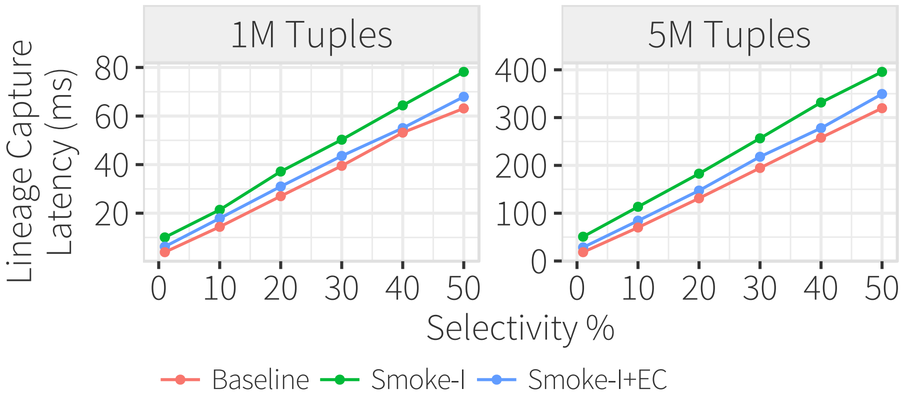


Figure 3.21: Latency of lineage capture techniques on selections with estimated selectivity (SMOKE-I-EC) and without (SMOKE-I). We find that it is better to overestimate than underestimate and incur resizing costs.

$50\%$. We evaluate SMOKE-I, as well as SMOKE-I-EC, which estimates the query selectivity as $\frac{v}{100}$ and, in turn, uses the selectivity estimates to preallocate the lineage indexes.

**Comparison of SMOKE techniques for selection.** SMOKE-I introduces average overhead of 0.38$\times$ and 0.46$\times$, for one and five million records across the varying selectivities. This is consistent with our finding that the techniques primarily vary by a constant per-tuple overhead. When using selectivity estimates, SMOKE-I-EC reduces the average overhead to 0.14$\times$ and 0.15$\times$, for the respective relation sizes. The reason that SMOKE-I-EC fluctuates is that the selectivity estimates may be slightly incorrect. When estimates overestimate the true selectivity, it is typically fine, however if they underestimate then they lead to array resizing overheads.

### 3.7.2 Multi-Operator Lineage Capture

To evaluate lineage capture on multi-operator plans, we used four queries from TPC-H (i.e., Q1, Q3, Q10, and Q12). Their physical query plans contain group by aggregation as the root operator, selections that vary in predicate complexity and selectivity, and up to three pk-fk joins. (Our hash-based execution precludes sort operations.) Figure 3.22 summarizes

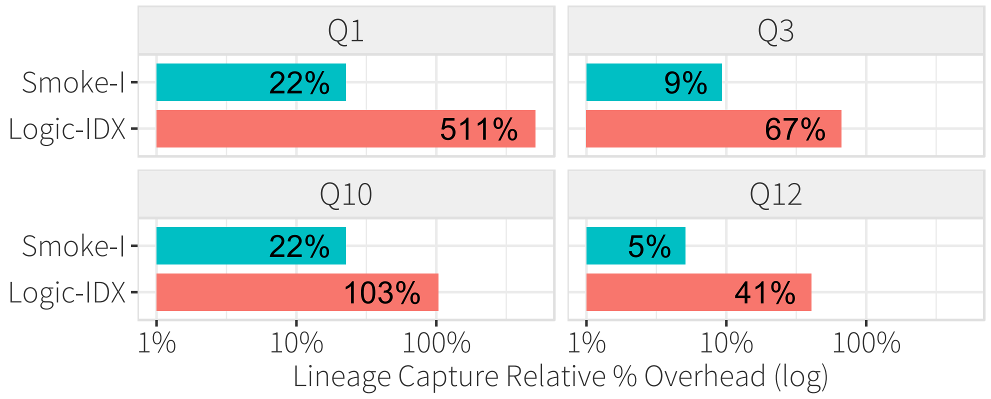


Figure 3.22: Relative overhead of SMOKE and logical lineage capture techniques for TPC-H queries Q1, Q3, Q10, and Q12. (SF=1)

the overhead of the best performing SMOKE (i.e., SMOKE-I) and logical (i.e., LOGIC-IDX) techniques for the four queries.

**Overall Results.** SMOKE-I reduces the capture overhead as compared to LOGIC-IDX by up to $22\times$. In addition, SMOKE-I incurs at most $22\%$ overhead across the four queries. To ensure that the reported overhead results are meaningful, we made sure that the query engine of SMOKE has reasonable performance. Despite its row-oriented execution, SMOKE is comparable to MonetDB (single-threaded, data cached in OS buffers): Q1 runs in $176$ms while the slowest query Q12 runs in $306$ms.[1] SMOKE-D (not shown) is slower than SMOKE-I due to the cost of $\bowtie_\gamma$ for lineage capture on the aggregation operator. However, it is still faster than the logical approaches. (We refer interested readers to our technical report [PW18] for a more detailed discussion.) Finally, although Q1 is simple (e.g., it has no joins), its results are arguably the most informative because its selections have the highest selectivity, which most stresses overheads as we discuss next.

**Impact of selections in lineage capture.** We found that the selectivity of the query predicate has a large impact on the overhead of logical approaches. Q1 introduces a setting where the predicate has a high selectivity. Thus, the input to the final aggregation operator has a

[1]The purpose is *not* to compare SMOKE with MonetDB, but to ensure that the reported overheads are over a reasonable baseline.

high cardinality. This leads output groups to depend on a large set of input records which, in turn, results in a large amount of duplication in the denormalized representation of the lineage graph. The other queries have low predicate selectivity which leads to lower (albeit significant) data redundancy. Overall, SMOKE is not sensitive to this effect because the lineage indexes represent the normalized lineage graph to avoid data duplication.

***Lineage Capture Takeaways (Sections 3.7.1 and 3.7.2).*** SMOKE-*based lineage capture techniques outperform both logical and physical alternatives by up to two orders of magnitude. Logical approaches that adhere to the relational model are affected by the denormalized lineage graph representation, extra indexing steps, and expensive joins. Physical approaches are affected by virtual function calls and write-inefficient lineage indexes. Array resizing contributes to a large portion of* SMOKE *overheads. However, accurate or overestimated statistics can further reduce resizing costs (up to 60%).*

### 3.7.3 Lineage Query Performance

We now evaluate the performance of different lineage query techniques. We evaluate the query: `SELECT * FROM` $L_b(o \in Q_{\mp}($`zipf`$),$ `zipf`$)$, where $Q_{\mp}(\text{zipf})$ is the query used in the group-by microbenchmark (Section 3.7.1.1) and $o$ denotes an output group. For this experiment, $Q_{\mp}($`zipf`$)$ contains 5000 groups while `zipf` contains $10M$ records and we vary its skew $\vartheta$. Varying $\vartheta$ highlights the query performance with respect to the cardinality of the backward lineage query. Figure 3.23 reports the lineage query latency for all $5000$ $o$ assignments and different $\vartheta$ values (i.e., $\vartheta \in \{0, 0.4, 0.8, 1.6\}$).

Recall that when we capture lineage with SMOKE-I; SMOKE-D; LOGIC-IDX; or PHYS-MEM, we evaluate lineage queries with SMOKE-L. SMOKE-L evaluates the lineage queries of our setup above using secondary index scans (i.e., it uses the contributing input rids of an output $o$ from the backward index of $Q_{\mp}$ to perform lookups into `zipf`). Next, we compare SMOKE-L with lazy, logical, and physical alternatives.

**Comparison with LAZY.** In contrast to SMOKE-L, LAZY performs a table scan of the input relation and evaluates an equality predicate on the integer group key. This is arguably

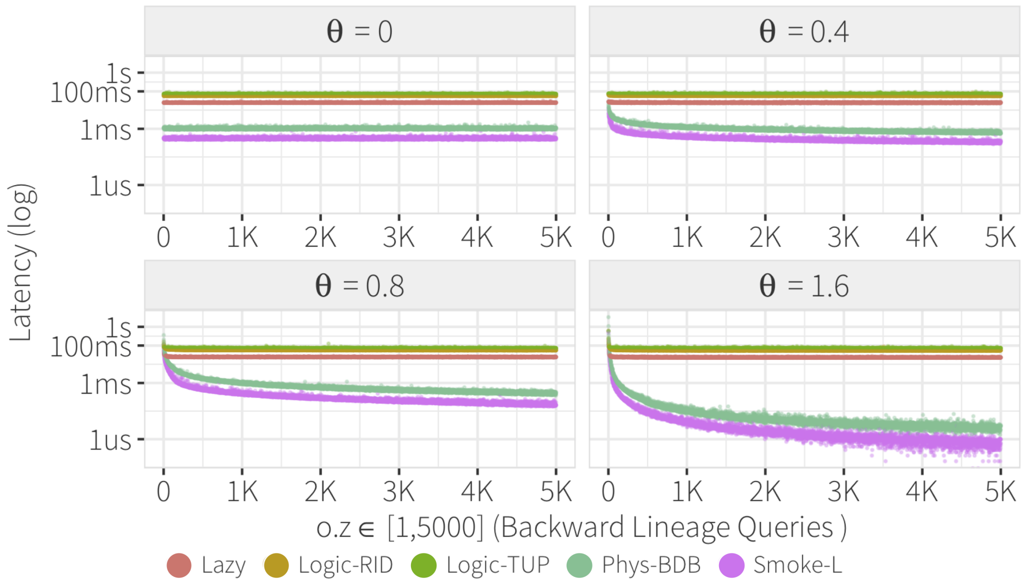


Figure 3.23: Lineage query latency for varying data skew ($\vartheta$). LAZY has a fixed cost to scan the input relation and evaluates a selection on the group-by key `o.z=?`. LOGIC-RID and LOGIC-TUP perform the same selection but on annotated output relations. SMOKE-L is mainly around $1\text{ms}$ and outperforms LAZY, LOGIC-RID, and LOGIC-TUP by up to five orders of magnitude for low selectivity lineage queries. The crossover points at high selectivities are due to the costs of SMOKE-L index scans. SMOKE-L is a lower bound for PHYS-BDB that incurs extra costs for reading from inefficient lineage indexes and communicating with external lineage subsystems.

the cheapest predicate to evaluate and constitutes a strong comparison baseline. We find that SMOKE-L outperforms LAZY up to five orders of magnitude, particularly when the cardinality of the output group is small. We expect the performance differences to grow when the base query uses more complex group-by keys, which increases the predicate evaluation cost, or when the input relation is wide, which increases scan costs [JRSS08; CGS03; KAI17]. Finally, there is a cross over point when the input relation is highly skewed ($\vartheta \in \{0.8, 1.6\}$) and the backward rid arrays of some groups have high cardinality. This increases the secondary index scan cost of SMOKE-L in comparison to the serial scan costs of LAZY, due to the multiple random memory accesses of the former.

**Comparison with logical techniques.** We also report the cost of scanning the annotated relations generated by LOGIC-RID and LOGIC-TUP (highest two lines). Scanning these relations to answer lineage queries is worse than LAZY because the annotated relation is wider than the input relation, yet they have the same cardinality. This is the main reason why we introduced extra indexing steps for the annotated output relations of logical approaches with LOGIC-IDX. (Recall that LOGIC-IDX is represented here by SMOKE-L.)

**Comparison with physical techniques.** PHYS-MEM is included as part of SMOKE-L, so we report PHYS-BDB. Using an external lineage subsystem to perform a lineage query, we need to perform function calls to the external system to fetch the input rids for an output group O. As long as we have the input rids, we can perform a secondary index scan to evaluate the lineage query similarly to SMOKE-L. In our experiments, we compared both fetching all input rids in a single function call as well as with consecutive function calls in a cursor-like fashion. The cursor-like approach outperformed the bulk approach since it avoids allocation costs for input rids. SMOKE-L provides a lower bound for PHYS-BDB: both perform the same secondary index scan but PHYS-BDB pays the cost of function calls to the external subsystem and it depends on indexes with worse read performance.

***Lineage Query Takeaways:*** SMOKE *outperforms logical and lazy lineage query evaluation strategies by multiple orders of magnitude, especially for low-selectivity lineage queries. We believe* SMOKE *is a lower bound for physical approaches by avoiding functions calls and using read-efficient indexes.*

## 3.8 Conclusions and Future Work

In this chapter, we showed how SMOKE performs efficient lineage capture on single- and multi-operator plans, under both INJECT and DEFER semantics, in a way that future backward and forward queries can be streamlined. Furthermore, we showed experimentally that SMOKE reduces the overhead of fine-grained provenance capture by avoiding shortcomings of logical and physical approaches in a principled manner. Moreover, on lineage query performance we showed that SMOKE improves on logical and physical approaches especially when backward (forward) queries have low selectivity on input (output) relations.

Going forth, there is ample space for future work primarily on reducing capture overheads and devising capture techniques in databases with designs other than the one of SMOKE.

Reducing further capture overheads, both latency- and memory-wise, is important provided that provenance is central to the optimization of many data-intensive applications, as we will see in Part II. In this direction, we believe that compressing lineage indexes following exact compression schemes (e.g., using roaring [CLKG16] or other well-known compression schemes [WLPS17]) or lossy ones (e.g., rid arrays of lineage indexes can be dropped or not materialized at all in cases when future provenance queries can be faster with serial scans instead of lineage-based indexed scans) can provide significant benefits both on memory consumption as well as latency overheads.

Injecting provenance capture within physical operators of engines that follow different designs (e.g., interpretation-based engines and compilation-based engines that do not follow the producer-consumer model); storage models (e.g., columnar representations and disk-based storage); and support, loosely speaking, query optimization strategies (e.g., vectorization, parallelization, or distributed execution) is another important direction for future work. While we expect the design principles that SMOKE embodies for provenance capture purposes to be universal, how each engine can embody them remains an open research question. For instance, introducing provenance capture techniques in an engine that performs vectorization requires revisiting the reuse principle (e.g., a vectorized selection typically generates a bitmap that maintains which records satisfy the selection—and such a bitmap can be realized as a lineage index) and the tight integration principle (e.g., vectorized engines need to perform lineage capture on a per batch and interleaved basis as opposed to the per tuple basis of SMOKE). As such, we believe that our design principles can serve as a guideline to reveal both optimization opportunities (e.g., bitmaps in vectorized selections) and potential limitations of SMOKE (e.g., batch and interleaved appends in our lineage indexes) when considering introducing provenance capture in alternative engines.

# Chapter 4

# Expressing and Evaluating Provenance Analytics

## 4.1 Introduction

In the previous chapter, we showed how to capture fine-grained provenance information over base queries involving individual relational operators as well as multi-operator plans. The end result of provenance capture is physical representations that map input to output records, and vice versa, based on the transformational provenance semantics. We also considered a simple provenance query model (i.e., backward and forward lineage queries) that allows applications to navigate between input and output records based on the generated mappings.

The logic of provenance consuming applications, however, may be complicated enough that exposing only this low-level query model could make application development a tedious process. In fact, traditional provenance-enabled systems have long proposed sophisticated provenance query models to either directly expose concrete provenance semantics (e.g., which [CWW00; GT17], why [BKT01], how [GKT07], and where [BKT01]) or more general purpose provenance query languages (e.g., ProQL [KIT10] and the query constructs of Ikeda et al. [Ike12]).

The introduction of sophisticated provenance query models overall highlight a class of analytics, that we refer to here as *provenance analytics*. Our focus in this chapter is to introduce techniques to express and evaluate provenance analytics based on the physical representation of provenance that our provenance capture techniques of Chapter 3 induce. (Performance-wise, we will show the benefits of our techniques in Chapter 5.)

To do so, we first revisit the data models induced by systems that follow the logical and physical provenance capture approaches, that we discussed in Section 3.2, to show that the representation that SMOKE provides for connections between input and output relations (i.e., rid indexes and rid arrays) are essentially physical representations of these data models. This is an important connection because it means that provenance query models introduced over these data models can also be directly expressed in SMOKE. The main difference is that the evaluation of provenance queries is subject to the physical representation of each system.

Having this connection in place, we then introduce techniques for the evaluation of several classes of provenance queries based on the induced physical representations by SMOKE. Our techniques cover the evaluation of general path queries, the evaluation of provenance consuming SQL queries which is a class that we first introduce here, and the evaluation of provenance semantics (i.e., which [CWW00; GT17], why [BKT01], how [GKT07], and where [BKT01] provenance).

## Contributions and Chapter Outline

In the rest of the chapter, we start with a necessary background and setup (Section 4.2). Then, we present our contributions as follows:

- We show that the provenance indexes that SMOKE provides as a product of provenance capture can be used as the physical representation for the data models induced by logical normalized, logical denormalized, and physical provenance capture approaches. As such, query models induced over these data models can be equally expressed in SMOKE, with the difference being that the evaluation of provenance queries is up to the physical representation of provenance by each system. (Section 4.3)

- We introduce techniques for the evaluation of path queries over provenance graphs that span multiple base queries by generalizing the notion of backward and forward queries over single base queries. (Section 4.4)

- We define the class of provenance consuming SQL queries (i.e., SQL queries that take as input the output of provenance path queries) and introduce techniques for their evaluation. (Section 4.5)

- We provide background and introduce techniques for the evaluation of which, why, how, and where provenance queries. (Section 4.6)

## 4.2 Background

For our discussion in this chapter, recall from Section 3.2 that a *base query* $\mathrm{Q}_{⏚}(\mathrm{D}) = \mathrm{V}$ is a relational query over a database of relations $\mathrm{D} = \{\mathrm{R}_1, \cdots, \mathrm{R}_\mathrm{n}\}$ that materializes a relation $\mathrm{V}$. An application can issue many base queries that we denoted as $\mathbb{Q}_{⏚} = \{\mathrm{Q}_{⏚1}, \cdots, \mathrm{Q}_{⏚\mathrm{m}}\}$. The result of executing multiple base queries is a set of materialized views $\mathcal{V} = \{\mathrm{V}_1, \ldots, \mathrm{V}_\mathrm{m}\}$. To account for a non-uniform naming of relations and views we refer to relations that pre-existed in a database as *base relations* and to relations that are products of base queries as *derivative relations*. Note that base queries and, in turn, derivative relations can be constructed by taking as input both base relations $\{\mathrm{R}_1, \cdots, \mathrm{R}_\mathrm{n}\}$ as well as other derivative relations $\mathcal{V} = \{\mathrm{V}_1, \ldots, \mathrm{V}_\mathrm{m}\}$. Finally, we consider every record of a relation to be uniquely identifiable through a row id (rid), as we also discussed in Chapter 3.

## 4.3 Data Models

In this section, we revisit the data models induced by logical and physical approaches as a result of provenance capture, to show that the physical representation that SMOKE provides can be regarded as a physical encoding for these data models.

### 4.3.1 Example Database

To illustrate the different definitions and ease our presentation, we consider a simple example database as shown in Figure 4.1. Relations $\mathrm{X}, \mathrm{Y}, \mathrm{Z}$ are base relations that pre-exist in the database. Relations $\mathrm{V}_1, \mathrm{V}_2, \mathrm{V}_3$ have been computed using the following base queries $\mathrm{V}_1 = \mathrm{Q}^1_{\perp}(\mathrm{X}, \mathrm{Y})$, $\mathrm{V}_2 = \mathrm{Q}^2_{\perp}(\mathrm{X}, \mathrm{Z})$, $\mathrm{V}_3 = \mathrm{Q}^3_{\perp}(\mathrm{Y}, \mathrm{Z}, \mathrm{V}_1)$. The exact contents of each relation and the exact base queries are of no use in our discussion and are not shown.

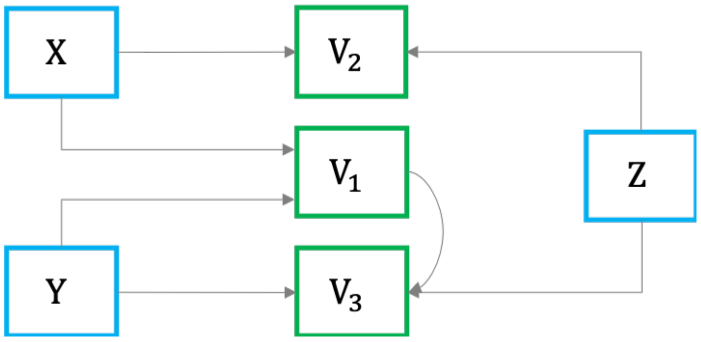


Figure 4.1: Example database that we use in our discussion. Relations $\mathrm{X}, \mathrm{Y}, \mathrm{Z}$ in blue boxes are base relations while $\mathrm{V}_1, \mathrm{V}_2, \mathrm{V}_3$ in green boxes are derivative relations.

### 4.3.2 Data Model of Logical Normalized Approaches

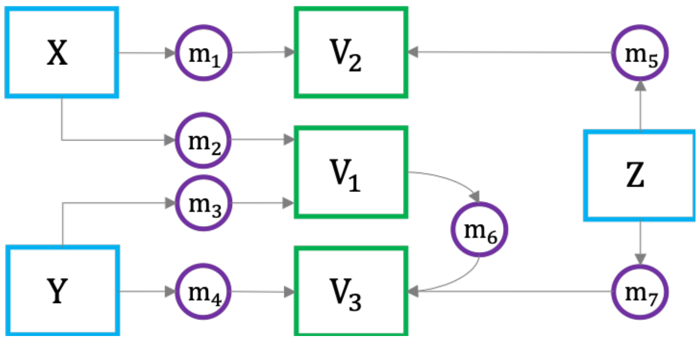


Figure 4.2: Data model generated by logical normalized approaches for our example database. Base and derivative relations are in blue and green boxes, respectively. Mapping relations connecting records between input and output relations are in purple circles.

In Chapter 3, we noted that systems, such as ProQL [KIT10] and Trio [ABS$^+$06], that follow logical normalized provenance capture approaches generate provenance relations that map input to output records and vice versa. Another way to put it, input and output relations are treated as dimensions while mapping relations are facts connecting tuples of the input and output dimensions. Considering multiple base queries over a database, their end data model is a graph $\mathrm{P(G, E)}$ over the database $\mathcal{D}$ which, for our example, is shown in Figure 4.2. This graph is constructed in the following way:

**Nodes** $\mathrm{G}$**.** Base and derivative relations, such as $\mathrm{X, Y, Z, V_1, V_2, V_3}$ of our example, are considered to be nodes in $\mathrm{G}$. Furthermore, for every pair of relations, say $\mathrm{(X, V_1)}$ where A is a base or derivative relation, B is a derivative relation, and $\mathrm{B}$ was constructed from a base query that involved $\mathrm{A}$, we create a *mapping relation*. A mapping relation $\mathrm{m_{AB}}$ has a conceptual schema `(arid, brid)` that maps which $\mathrm{A}$ record, indicated by its rid `arid`, contributed to which $\mathrm{B}$ record, indicated by its rid `brid`. In our example, there are seven such mapping relations shown in purple circles in Figure 4.2. As an example, $\mathrm{m_1}$ keeps track of the rid connections between the records of $\mathrm{X}$ and $\mathrm{V_1}$ since $\mathrm{X}$ contributes to $\mathrm{V_1}$. We refer to this rid-based schema of a mapping relation as a conceptual one because provenance applications may extend it with more attributes for their own application logic. Finally, note that mapping relations are also considered nodes in $\mathrm{G}$.

**Edges** $\mathrm{E}$**.** For every mapping relation $\mathrm{m_{AB}}$ that connects two relations $\mathrm{(A, B)}$ we introduce directed edges $\mathrm{(A, m_{AB})}$ and $\mathrm{(m_{AB}, B)}$ in the graph $\mathrm{P}$. The direction of the edges denotes which relation was input to base queries and which relation was the derivative. A different way to see this is that mapping relations annotate edges between relations with edges denoting workflow. To conform with actual data models of logical normalized approaches, however, we consider edges between relations to be split through mapping relations.

**Connection with SMOKE.** The end result of the provenance capture that we introduced in Chapter 3 is a graph that connects inputs to output tuples with respect to the semantics of a query. This graph is precisely the encoding of SMOKE for the mapping relations of the graph $\mathrm{P(G, E)}$ that we defined above. Hence, the provenance graph $\mathrm{P(G, E)}$ is a generalization of the fine-grained provenance graph, that we introduced in Section 3.2, to account for provenance capture across many base queries and also introduces naming for mapping relations so that they can be used by query models. Furthermore, it is important to note that the definition of the mapping relations that we discussed above is only logical. The actual physical representation in SMOKE of mapping relations is our fine-grained provenance capture indexes (i.e., rid indexes and rid arrays) that we introduced in Chapter 3, and our overall discussion in the previous chapter was on how to generate these mappings efficiently.

### 4.3.3 Data Model of Logical Denormalized Approaches

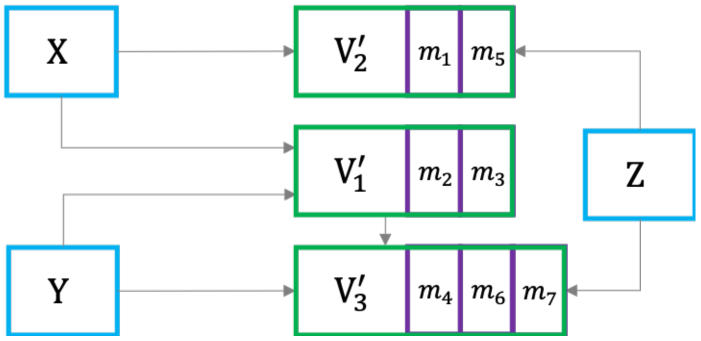


Figure 4.3: Data model generated by logical denormalized approaches for our example database. Base and derivative relations are in blue and green boxes, respectively. Mapping relations connecting records between input and output relations are shown in purple and are part of derivative relations.

In contrast to normalized approaches, denormalized provenance capture approaches, such as Perm [GA09] and its ancestor GPROM [NKG+17], store the mappings within output relations, as we noted in Chapter 3. Again considering multiple base queries, the end result of provenance capture is a data model where all derivative relations are annotated with mappings. For our example, the end result is shown in Figure 4.3. Mappings (shown in purple) are part of the output relations. As such, in contrast to the normalized approaches, the mapping information should only encode information from the input.

In general, denormalized approaches are more suited for ad-hoc provenance analytics where a user issues a base query and, as a result, retrieves the result of the query annotated with provenance information. Provenance analysis, which is concerned with the retrospective analysis of provenance graphs, can still be extracted from denormalized representations. This is possible by either converting the mappings to their normalized representation as a post-processing step or by expressing provenance queries as relational queries over the derivative denormalized relations. The latter approach is considered heavy due to the redundancy incurred by the derivative denormalized representations to encode the graph-based provenance information, as we also showed experimentally in Chapter 3. As such, provenance systems typically follow the former approach of converting to normalized representations before performing provenance analytics. Hence, next we will show the connection of SMOKE with the data model of logical denormalized approaches, to cover query models that issue relational queries over denormalized representations, and in the remainder of this chapter we will focus primarily on query models over normalized representations.

**Connection with SMOKE.** Consider again the way we physically store backward rid indexes and rid arrays. For rid indexes, each entry maintains an rid array that stores the input rids and this entry is aligned in memory with the output entry that the inputs contribute to. Similar is the case for rid arrays. Putting it differently, consider relations stored as column stores. SMOKE's rid arrays and indexes can be considered individual columns in this representation. Then, the end result is the same data model with the result of the denormalized approaches modulo three differences at the physical representation: First, rid indexes allow us to avoid the denormalization effect of output relations (i.e., each output entry with $k$ inputs contributing to it needs to be replicated $k\times$ in denormalized approaches. SMOKE avoids the denormalization by appending all $k$ rids in an array that is associated with the output entry due to the construction of our rid indexes). In other words, SMOKE uses a nested relational encoding for annotations of output records to avoid the denormalization effect. Second, SMOKE can also annotate input relations with mappings to accelerate queries involving forward tracing which is not possible by denormalization approaches. (This is because an input relation cannot be annotated as a result of a base query.) Finally, no matter whether the underlying representation of input and output relations, SMOKE introduces mappings as separate columns. This means that even if an input or output relation is a row-store, access to mappings is columnar. This is important to avoid the effect of scanning wide tables that hurts the provenance query performance, as we showed in Section 3.7.

### 4.3.4 Data Models of Physical Approaches

Finally, we also briefly outline the connections of SMOKE with data models of physical approaches. Recall that physical approaches store the connections between input and output relations on a separate subsystem that is responsible for their physical representation. Since provenance information is graph-based, physical approaches typically store provenance in graph representations. The query models for such approaches follow the one of key-value stores. That is, given one or more rids of output (input) records they return the input (output) rids that contributed to the given output (input) rids. Since SMOKE is part of this class, the evaluation of provenance queries that we discuss in this chapter can also be used by the other systems. Our contribution is to show that queries expressed over the data models of logical approaches can also be expressed by SMOKE. As such, we expect other proposed physical approaches to follow similar evaluation techniques. Note, however, that some physical approaches [IST$^+$15; LDY13] encode provenance using relational representations that we discussed above. Such approaches should instead follow the evaluation techniques as proposed for logical approaches or change their representation to the one of SMOKE and use the techniques that we propose here.

Having defined the provenance graph $\mathrm{P}(\mathrm{G}, \mathrm{E})$ and its semantics as well as the connections of SMOKE with the data models of alternative provenance capture approaches, we next proceed to discuss different query models and their evaluation in SMOKE.

## 4.4 Path Queries

In Chapter 3, we considered a simple query model of backward (forward) queries $\mathrm{L_b}$ ($\mathrm{L_f}$) that take as input a subset of output (input) tuples and return the subset of input (output) tuples that contributed to (were contributed by) the given subset of output (input) tuples. Since in this chapter we consider multiple base queries, it is natural to ask for backward and forward traces that span multiple base queries.

**Path queries.** A path query over a database with base and derivative relations can be composed in two ways. First, if there is no ambiguity in terms of what path connects two relations $V_i$ and $V_j$, then one can specify a path query `trace(`$o \in V_j$`, `$V_i$`)` that traces the subset $o \in V_j$ to $V_j$. If there are many paths connecting the two relations $V_i$ and $V_j$, however, then this trace query is ambiguous. In such cases, the full path specification should be provided `trace(`$o \in V_j$`, [`$m_{jl}, V_l, \ldots, V_k, m_{ki}, V_i$`])`, where $m_{xy}$ refers to the mapping relations that we introduced in Section 4.3. (Note that this ambiguity was not a problem for $L_b$ and $L_f$ because $L_b$ and $L_f$ were specified with respect to a single base query and input relations were referenced by instance and not by name as is the case here.) Finally, for completeness, we note that the subset $o \in V_j$ can be specified either by specifying the rids of the records $o \in V_j$, by relational selections on $V_j$, or even by trace queries to $V_j$.

**Evaluation of path queries in SMOKE.** To evaluate the path queries that we introduced above SMOKE exploits the transitivity property induced over the mappings. Consider our example data model in Figure 4.2. $V_2$ has been constructed from a base query $Q^1_{\mp}(X, Z)$. In turn, $V_3$ has been constructed from a base query $Q^3_{\mp}(Y, V2)$. A trace query can ask for the records in X that contribute to a specific subset of records in $V_3$, although $V_3$ has not been constructed by taking as input X. Backward indexes connecting $V_3$ to $V_2$ can help us get from a subset of the output of $V_3$ to the subset of $V_2$ that contributed to the given subset of $V_3$. To do so, we use the backward indexes connecting $V_2$ to X to get the subset of X that contributed to the subset of $V_2$ that, in turn, we got by backward tracing from $V_3$ to $V_2$. Hence, SMOKE simply evaluates path queries by recursively evaluating backward and forward queries based on the transitivity over provenance graphs of multiple base queries.

## 4.5 Provenance Consuming SQL queries

Provenance consuming applications (e.g., interactive data visualizations or interactive data profiling) rarely use provenance information in its raw form (i.e., the output of provenance path queries). Rather they want to transform the provenance information in a way that is meaningful for their own application logic.

To this end, provenance information can be consumed and processed using the full analytical power of SQL. This is possible due to the main observation that the output of path queries are subsets of relations. Hence, SQL queries can take as input and process the output of provenance path queries and provide powerful analytical capabilities to end users. Next, we show how SQL queries that take as input the output of provenance path queries can be expressed and evaluated by SMOKE in an ad-hoc manner. Note, however, that in Section 5.4 we will introduce optimizations to push the SQL consuming logic into the provenance capture phase if such queries are known when we perform provenance capture. This is typically the case for provenance applications with fixed (exact or parametrized) logic such as interactive data visualizations or profiling.

Now, provenance consuming SQL queries can be naturally expressed by specifying path queries in the FROM clauses of SQL queries. For instance, suppose that we want to count the number of records of X that contributed to a given subset $o \in V_3$ of our example in Section 4.4. We can express that in the following way:

```
SELECT COUNT(*) FROM trace(o ∈ V3, X)
```

The output of the trace query is the subset of X that contributed to $o \in V_3$, and the COUNT aggregate just counts the number of records in this particular subset.

**Evaluation in SMOKE.** The evaluation strategy of provenance consuming SQL queries in SMOKE is straightforward. Path queries result in index scans of relations that parent physical operators can use to implement the logic of the specified SQL queries.

## 4.6 Provenance Semantics

Provenance semantics, also referred to as provenance types, regard interpretations of the ways an output record has come into existence. Major provenance semantics involve which-, why-, how-, and where-provenance which are concerned with which minimal set of records contributed an output result, why an output result has come into being, how an output result has come into existence, and from where in the input has an attribute value of an output record gotten its value from, respectively.

How-provenance has been one of the most major classes of provenance semantics since it can encode several other provenance semantics, including which and why, as well as used to annotate output results with semiring-based values. Where-provenance queries cannot be directly answered based on how-provenance alone but they require how-provenance information to be computed.

In this section, we will gradually show how SMOKE can evaluate which-, why-, and how-provenance queries. Based on the evaluation of how queries we will also show how we can perform annotation propagation by the evaluation of semirings and, in turn, how to evaluate the general class of ProQL [KIT10] type of queries based on the annotation propagation and the provenance path queries of Section 4.4. We conclude this section by showing how SMOKE can evaluate where-provenance queries.

### 4.6.1 Which-Provenance

Which-provenance [GT17], first introduced by Cui et al. [CWW00] as data lineage, describes the maximal set of records from each input relation that contributes to an output. The definition of which-provenance, paraphrased from [CCT09; CWW00], is as follows:

---

**Definition 1 (Which-provenance (or data lineage) for a relational operator)** Let $\mathrm{Op}$ be any relational operator over relations $\mathrm{R_1, \dots, R_n}$. The which-provenance of a record $\mathrm{t \in Op(R_1, \dots, R_n)}$ is a sequence $\mathrm{\langle R'_1, \dots, R'_n \rangle}$ of subsets $\mathrm{R'_i \subseteq R_i}$ s.t.:

[1] $\mathrm{Op(R'_1, \dots, R'_n) = \{t\}}$

[2] $\mathrm{\forall i \in [1, n]}$ and $\mathrm{\forall r_i \in R'_i}$ we have $\mathrm{Op(R'_1, \dots, R'_{i-1}, \{r_i\}, R'_{i+1}, \dots, R'_n) \neq \emptyset}$

[3] $\mathrm{\langle R'_1, \dots, R'_n \rangle}$ is maximal among subsets of $\mathrm{R_1, \dots, R_n}$ satisfying [1] and [2].

---

Intuitively, as is also noted in [CCT09; CWW00], condition [1] ensures that which-provenance is relevant to tuple $\mathrm{t}$. Condition [2] ensures that no "irrelevant" records are included in the which-provenance of $\mathrm{t}$ and that every record $\mathrm{r_i}$ in every subset $\mathrm{R'_i}$ in the which-provenance of $\mathrm{t}$ contributes something to $\mathrm{t}$. Finally, condition [3] ensures that which-provenance contains exactly all the tuples that contribute to $\mathrm{t}$.

Based on this general definition of which-provenance, Cui et al. [CWW00; Cui01] proposed definitions of which-provenance queries over individual relational operators. The definition of which-provenance for individual operators is exactly the same with the result of backward provenance queries that we introduced in Chapter 3 for transformational provenance semantics. The main difference between transformational and which-provenance is on some types of multi-operator plans under which an input record $\mathrm{r}$ contributes multiple times to an output record $\mathrm{t}$. In such cases, which-provenance will return only one instance of $\mathrm{r}$, to ensure the conditions of Definition 1, as opposed to returning $\mathrm{r}$ as many times as it contributed to $\mathrm{t}$ (that backward queries over transformational provenance would return).

To make this difference more clear, consider the following base query:

```
Q⏚=SELECT   COUNT(*), X.cname, Y.pname
   FROM     X.cid = Y.cid
   GROUP BY X.cname, Y.pname
```

Furthermore, assume that we execute $\mathrm{Q}_{⏚}$ over the following database instance (tables `X` is on the left side and table `Y` is on the right side):

| | cid | cname |
|---|---|---|
| $x_1$ | 1 | Bob |
| $x_2$ | 2 | Alice |

| | oid | cid | pname | date |
|---|---|---|---|---|
| $y_1$ | 1 | 1 | iPhone | 12/25 |
| $y_2$ | 2 | 1 | iPhone | 12/25 |
| $y_3$ | 3 | 2 | XBox | 12/25 |

The output of $Q_{\underline{\perp}}$ over the above database instance is the following:

| | COUNT(*) | X.cname | Y.pname |
|---|---|---|---|
| $o_1$ | 2 | Bob | iPhone |
| $o_2$ | 1 | Alice | xBox |

According to the transformational provenance semantics, and their corresponding physical representation in SMOKE, the backward index for $o_1$ with respect to table $X$ contains the rid $x_1$ *twice*. As a result, a backward query from the output of $Q_{\underline{\perp}}$ to X will output the record with rid $x_1$ twice. In contrast, which-provenance semantics, as imposed by Definition 1, requires us to return $x_1$ only once in response to which-provenance queries. This can be important for applications that do not care on how many times an input record has contributed to an output record but rather just *which* record contributed to a result.

## Evaluation in SMOKE

To evaluate which-provenance queries SMOKE, instead of just returning the input rids for a given output per relation, it first performs, if required by the structure of the query, a de-duplication of rids per rid array in backward rid lists, as is required by Definition 1. That is, a which-provenance query $\mathrm{Which}(o_1)$ of our example will return the sequence $\langle\{x_1\},\{y_1,y_2\}\rangle$ instead of $\langle\{x_1,x_1\},\{y_1,y_2\}\rangle$ which would be the result of backward queries with transformational provenance. Note that in our discussion above we only considered which-provenance queries with respect to a single base query. A generalization for which queries that span across multiple base queries is straightforward since we can still take the output of the which query over a single base query and then recursively perform which queries over other base queries.

### 4.6.2 Why-provenance

Which-provenance, as discussed above, returns the multiset of records for each input relation that contributed to a given output result. This multiset however does not provide an explanation of why the output record has come into existence by means of how the different input records have been combined to provide this result.

To account for this lack of semantics, Buneman et al. [BKT01] introduced the notion of why-provenance that encodes the connections between input records that an output record depends on. To understand the main difference between which- and why-provenance, consider again the example that we gave in Section 4.6.1. A why-provenance query Why($o_1$) will return the multiset $\{(x_1, y_1), (x_1, y_2)\}$ as opposed to $\langle\{x_1\}, \{y_1, y_2\}\rangle$ that Which($o_1$) would return. Intuitively, why-provenance tells us not just which records contributed to a particular result but also what combinations of input records witness the output. Hence, the results of why-provenance queries are also called witnesses of output. That is, for our example, the connections $(x_1, y_1)$ and $(x_2, y_2)$ witness the existence of the output $o_1$.

#### Evaluation in SMOKE

SMOKE answers why-provenance queries by exploiting the alignment property of backward index rid lists with the output record and with each other. Consider again the output of $Q_{\perp}$ in our example this time with the backward index rids $\texttt{bw}_X$ and $\texttt{bw}_Y$:

| | **COUNT(*)** | **X.cname** | **Y.pname** |
|---|---|---|---|
| $o_1$ | 1 | Bob | iPhone |
| $o_2$ | 2 | Alice | xBox |

| $\texttt{bw}_X$ |
|---|
| $x_1, x_1$ |
| $x_2$ |

| $\texttt{bw}_Y$ |
|---|
| $y_1, y_2$ |
| $y_3$ |

To evaluate why-provenance queries in SMOKE we concatenate the rids based on their positions. That is for a why- provenance query for $o_1$ the first $x_1$ of $\texttt{bw}_X$ will be concatenated with $y_1$ of $\texttt{bw}_Y$ while the second $x_1$ will be concatenated with $x_2$. The reason why this works is because when we perform provenance capture the records that contributed to a specific output from different input relations will be appended at the same positions of the different rid lists. To generalize on answering why-provenance queries across base queries we can recursively apply why-provenance queries on the individual rids in the result of previous why-provenance queries.

### 4.6.3 How-provenance

Both which- and why-provenance that we discussed above provide us with sets of input records to explain the existence of a given output. However, they do not encode *how* an output record has come into being. In other terms, they do not encode how input records were processed due to the semantics of the base query to contribute to the output.

To this end, Green et al. [GKT07] introduced the notion of how-provenance that encodes how input records where combined to provide a result due to the semantics of a query besides only providing set of records that contributed to a result. Next, we provide the necessary intuition behind how-provenance and its connections with polynomials and semirings. For their theoretical grounds, interesting readers are referred to [GKT07; CCT09].

The main idea behind how-provenance is grounded on the fact that relational operators can either combine records (e.g., join) or merge (e.g., set projection, grouping, or set union) input records together. Whenever we combine input records together, how-provenance encodes the connection as an *abstract product*, denoted with $\star$. When we merge them, we encode the connection with an *abstract sum*, denoted with $+$. In our example, the how-provenance for the record $\mathrm{o}_1$ is $(\mathrm{x}_1 \star \mathrm{y}_1) + (\mathrm{x}_1 \star \mathrm{y}_2)$ because $(\mathrm{x}_1, \mathrm{y}_1)$ and $(\mathrm{x}_1, \mathrm{y}_2)$ were combined and then merged as the result of the join and grouping operators. The end result of how-provenance is that each output record is now associated with a polynomial, due to combining $\star$ and merging $+$ input records. This provenance polynomial explains how the output came into being with respect to the semantics of a base query.

Perhaps more interestingly, provenance polynomials do not only tell us how an output record was derived, however. Rather they also allow us to annotate output records by evaluating the polynomials under different definitions of the abstract sum, abstract product, and base values. While above we provided some intuition behind the notions of abstract sum and product, we also need to clarify the notion of base values. So far, we have considered that input records are only associated with rids. In general, however, we can also consider input records to be annotated with any value that follows the semantics of an application. These are referred to as base values that annotate input records.

Now, if we assume that base values are drawn from a specific domain and we have a definition of the abstract product and sum for values over this domain then evaluating polynomials of how-provenance is equivalent to evaluating polynomials of a commutative *semiring*. Recall that semirings are mathematical objects $(\mathrm{K}, 0, 1, +, \star)$ with $+$ and $\star$ denoting the abstract sum and product operations over elements drawn from the domain $\mathrm{K}$. $0$ and $1$ are drawn from $\mathrm{K}$ and correspond to identity elements for $+$ and $\star$, respectively.

As a concrete example, consider the provenance polynomial $(\mathrm{x}_1 \star \mathrm{y}_1) + (\mathrm{x}_1 \star \mathrm{y}_2)$ for the output record $\mathrm{o}_1$ above. Instead of rids, consider that we have annotated input records with `false` and `true` values, say, `true` for $\mathrm{x}_1$, `false` for $\mathrm{y}_1$, and `true` for $\mathrm{y}_2$. Such base values could denote whether we trust a record or not. Furthermore, assume that the abstract sum and product correspond to the logical and $\wedge$ and logical or $\vee$ operations, respectively. Then, the provenance polynomial becomes $(\mathrm{x}_1 \vee \mathrm{y}_1) \wedge (\mathrm{x}_1 \vee \mathrm{y}_2)$ which evaluates to true and could denote that we should trust the output record.

As a result of the semiring construction and the semantics it can expose to provenance consuming applications, several semirings have been proposed each exposing different semantics. In fact, our example above demonstrated the *trust* semiring that is typically used in the context of collaborative systems (i.e., different peers annotate records with true or false denoting whether they trust them or not and, based on these annotations, provenance systems should infer if output records derived from the annotated records should be trusted or not). Interested readers are referred to [KIT10; CCT09] for the definitions of other important semirings, including derivability; confidentiality; lineage; and probability semirings among others; and real-world use cases behind them.

### Evaluation in SMOKE

SMOKE evaluates how-provenance queries and semirings similarly to how ORCHESTRA computes semirings in response to PROQL queries. In our discussion next, we first show the equivalence between the two systems in terms of physical representations of mappings and overall data models. This implies that SMOKE can directly borrow the evaluation strategies of ORCHESTRA for PROQL queries. As such, we only briefly discuss how SMOKE evaluates semirings and we refer interested readers to a discussion over evaluation strategies in [KIT10].

PROQL is a provenance query language that allows end users to specify semiring computations over the provenance graph captured within ORCHESTRA. ORCHESTRA captures provenance with logical normalized approaches and its data model of the provenance graph is the one we introduced in Section 4.3.2 (modulo that mappings in ORCHESTRA do not store rids but records are identified with primary keys.) This data model has poor performance when it comes to evaluating path queries since mappings need to be joined. Hence, Karvounarakis et al. [KIT10] index the mappings generated by ORCHESTRA in access support relation (ASR) indexes. Recall that we referred to this class of techniques (i.e., provenance capture in relational forms followed by indexing of provenance) as LOGIC-IDX and showed its shortcomings for provenance capture in Section 3.7. Regardless of the provenance capture, however, ORCHESTRA and SMOKE end up exposing the same data models, as we showed in Section 4.3.2, and similar underlying physical representations since SMOKE's provenance indexes are equivalent to the ASR indexes of ORCHESTRA. As such, SMOKE and ORCHESTRA respond in the same ways to PROQL queries.

To compute a semiring in SMOKE, one first needs to annotate records with base values. This can be done by either introducing more attributes or with separate relations with one to one connection with the relations to be annotated. Then, given the definition of the abstract sum and product operations of the semiring, SMOKE computes the provenance polynomials for output records by using knowledge of the query and the backward rids. In our example, knowing that records are joined and then grouped provides us with the necessary information to perform an inner product of the rids of the backward indexes and derive the provenance polynomial $(\mathrm{x}_1 \star \mathrm{y}_1) + (\mathrm{x}_1 \star \mathrm{y}_2)$. Knowing how to construct the provenance polynomial for each output record, we can now propagate base values, compute the semiring, and perform the annotation of the output records. Note that this construction is for computing how-provenance only for a single base query. However, the same construction follows in a bottom-up fashion when we perform semiring evaluations across base queries. We can do so because evaluating a semiring over a single base query results in annotating its outputs with values. In turn, these values are treated as base values for semiring evaluations over base queries that take as input the so far semiring-based annotated outputs. For a more detailed discussion on this bottom-up construction interested readers are referred to [KIT10]. Finally, note that this construction imposes that provenance graphs that both SMOKE and ORCHESTRA account for are acyclic; otherwise this construction will fail. Accounting for cyclic provenance graphs is an interesting problem for future work.

### 4.6.4 Where-provenance

So far, we have discussed how different provenance types encode input records to explain outputs. These explanations, however, are at the whole record level. In contrast, a natural question to ask is from where does a given attribute value in an output has "copied" its value from. Similarly, we can ask to what output attribute value has a given input attribute value contributed its value. Such types of questions are called where-provenance queries and encoded through the notion of where-provenance [BKT01; CCT09].

A $\mathrm{Where}$ provenance query takes as input an attribute value and, instead of record encodings that we show with the previous provenance types, returns sets of *locations* where a location is a triple *(relation, record, attribute name)*, to answer from what relation, which record, and from what attribute has the attribute value been derived from.

### Evaluation in SMOKE

Evaluation of where queries in SMOKE is straightforward given that path queries can take us back to the input that produced the particular output. Then, the only thing that remains is to find what attribute(s) from the traced inputs has contributed to the given attribute. This information is encoded in the base query that produced the output of interest. Hence, SMOKE evaluates where-provenance queries by first analyzing the query plans of base queries to identify what input attribute contributed to which output attribute and then using provenance information to trace back from the given output to the inputs that contributed to the output. In case the analysis of query plans shows that there are no input attributes that can contribute to the given output attribute then SMOKE responds with `undefined`.

## 4.7 Conclusions

In this chapter, we showed how SMOKE expresses and evaluates analytical provenance queries including path queries, provenance consuming SQL queries, and several of the established provenance semantics. Furthermore, we showed that the physical representation of provenance provided by SMOKE provides a physical representation of the data models induced by alternative logical and physical provenance capture approaches. As such, our main conclusion is that SMOKE is as expressive as other state-of-the-art provenance enabled systems for the class of queries we considered here. The performance of our techniques in comparison with state-of-the-art alternatives will be presented in Chapter 5. We conclude our discussion by noting that several other classes of analytical provenance queries are not currently handled by SMOKE including provenance semantics beyond SPJAU queries and provenance propagation over acyclic graphs which are interesting future work.

# Chapter 5

# Optimization of Provenance Analytics

## 5.1 Introduction

In the previous chapter, we considered expressing and evaluating provenance analytics over provenance data models as induced by the provenance capture phase. This enables provenance consuming applications to evaluate provenance queries in an ad-hoc manner. However, many provenance applications, such as those in interactive visualization; profiling; or security, may have a pre-defined provenance consuming logic that amounts to a pre-declared provenance consuming query workload W. Knowing this query workload at the moment of the provenance capture phase enables several optimizations on the provenance capture phase with the goal to streamline future provenance queries in W.

To this end, in this chapter, we introduce several simple yet effective optimizations that exploit knowledge of W to avoid capturing provenance information and generate physical representations that directly speed up queries in W. Overall, we have designed our optimization techniques based on the following three design principles **P4-P6**. (Note that our design principles here build on the principles **P1-P3** that we described in Section 3.1.)

**P4. Capture avoidance.** Provenance applications such as debugging need to capture provenance to answer ad-hoc provenance queries that can trace back or forth to any input, intermediate, or output table. For applications such as interactive visualizations or profiling, however, provenance queries may be known up-front. SMOKE uses this apriori knowledge to avoid materializing provenance that will not be queried in the future.

**P5. Pre-computing provenance consuming SQL results.** Provenance applications rarely require all results of a provenance query (e.g., all records that contributed to an aggregation result) unless the results have low cardinality. Instead, the results are filtered, transformed, and aggregated by additional SQL queries. We termed these queries *provenance consuming SQL queries* in Chapter 4. If such queries are known up-front, as is typically the case for applications with templated analysis capabilities (e.g., Tableau or Power BI), SMOKE pushes physical design optimizations into the provenance capture phase. These optimizations are used to speed up future provenance consuming queries, and can include provenance index partitioning, materializing aggregates, or collecting statistics.

**P6. Pre-computing provenance semantics.** Similarly to the provenance consumption using SQL, provenance applications may want to operate under only which, why, or how provenance semantics as opposed to the transformational provenance semantics that SMOKE provides from its provenance capture. As such, instead of capturing transformational provenance and evaluate retrospectively other provenance semantics, we present techniques that derive the desired provenance semantics directly at the provenance capture phase.

## Contributions and Chapter Outline

In the rest of the chapter, we start by providing some necessary setup to ease our overall discussion (Section 5.2). Then, we present our contributions as follows:

- We introduce techniques that avoid capturing provenance, either for individual relations or directions, that will not be used by future provenance queries. (Section 5.3)

- We introduce techniques that push down the logic of future provenance consuming SQL queries down into the provenance capture phase. Our techniques generate novel physical representations that streamline such future provenance queries. (Section 5.4)
- We introduce techniques that pre-compute provenance semantics. (Section 5.5).
- Experimentally, we compare our optimizations both with the techniques for ad-hoc evaluation of provenance queries that we presented in Chapter 4 as well as with known, state-of-the-art alternative techniques. Our experimental analysis includes results both for the provenance capture and provenance query phases. (Section 5.6).

## 5.2 Setup

To ease our discussion throughout this chapter, we present optimizations on the provenance capture phase for the following simple base query:

$$Q_{\mp} = \sigma_{\texttt{o\_orderdate>`2017-08-01'}}(\texttt{orders} \bowtie \texttt{lineitem})$$

$Q_{\mp}$ joins `orders` with `lineitem` records and selects only the join results having `o_orderdate>‘2017-08-01’`. Furthermore, assume that the `orders` table has the following schema: `orders(oid, o_orderdate, o_shipdate)`. The schema for the `lineitem` is irrelevant to our discussion.

The optimization techniques that we introduce throughout this chapter are workload-aware. Hence, we expect a given future workload W synthesized out of analytical provenance queries. Each optimization technique targets different types of analytical provenance queries that may reside in W. As such, we present examples of such provenance queries in-line per optimization technique. To further motivate our techniques, we draw such examples from the domain of interactive data visualizations.

## 5.3 Provenance Pruning

Our pruning optimizations disable provenance capture for provenance indexes that will not be used in W. We present two types of pruning that disable provenance capture for specific input relations and provenance directions (i.e., backward or forward).

### 5.3.1 Pruning Input Relations

A simple visualization of $Q_{\perp}$ could show a tooltip of `lineitem` information when a user hovers over a visualized result of $Q_{\perp}$. As we will see in Chapter 7, this functionality can be expressed as a backward query from an output of $Q_{\perp}$ to `lineitem` to fetch the lineitem record responsible for the hovered output. Assuming that there are no other interactions involving provenance queries on the `orders` table, it is clear that the provenance indexes for the `orders` table will not be used in any way. As such, SMOKE changes the provenance capture techniques that we presented in Chapter 3, to avoid capturing provenance for `orders` in $Q_{\perp}$. In general, SMOKE does not capture provenance for relations not referenced in the future workload W, and this is possible by not integrating the provenance capture logic within the physical plan of $Q_{\perp}$ for tables not referenced in W.

### 5.3.2 Pruning Provenance Directions

Extending the previous example, it is clear that W will only execute a backward provenance query to `lineitem` and not vice versa. Thus, SMOKE can also avoid generating the forward provenance index from `lineitem` to the base query output because it will not be used in any way from the analytical provenance queries in W. The provenance indexes that can be pruned are evident from the provenance consuming queries in W. To do avoid capturing provenance directions, SMOKE does not integrate the provenance capture logic within the physical plan of $Q_{\perp}$ that is responsible for their construction.

## 5.4 Provenance Consuming SQL queries

User-facing applications rarely present a large set of query results to a user. Instead, they *reduce* the result cardinality with further filter, transformation, and aggregation operations. These reductions can be expressed as provenance consuming SQL queries, that we introduced in Section 4.5. Here, we show how such consumption logic can be pushed into the provenance capture logic if provenance consuming SQL queries are available in $\mathrm{W}$. More specifically, we next present three simple, yet effective, push-down optimizations for fixed and parametrized predicates as well as group-by aggregations.

### 5.4.1 Selection Push-down

Visualizations often update metrics that summarize data based on user selections. For instance, the following query retrieves Christmas shipment order information for parts of the visualization that the user interacts with: $\mathrm{C} = \sigma_{\mathrm{shipdate}=\text{`xmas'}}(\mathrm{L_B}(\mathrm{O}' \subseteq \mathrm{Q}_{\mp}(\mathrm{D}), \mathrm{orders}))$. Our selection push-down optimization pushes down the predicate `shipdate='xmas'` into provenance capture, so that SMOKE will first check whether the input tuple satisfies the predicate before adding it to the provenance indexes. If the predicate is on a group-by key, SMOKE does not capture provenance for all other groups. This reduces provenance space overheads and usually reduces capture overheads. If the predicate is expensive to evaluate (e.g., slow UDF), it can increase capture overheads.

### 5.4.2 Data Skipping Push-down

Pushing down selections requires fixed predicates. However, interactive visualizations also use parametrized predicates. For instance, a user may use a slider to dynamically change the shipping date (`:p1`): $\mathrm{C} = \sigma_{\mathit{shipdate}=:\mathrm{p1}}(\mathrm{L_B}(\mathrm{O}' \subseteq \mathrm{Q}_{\mp}(\mathrm{D}), \mathrm{orders}))$. This pattern is ubiquitous in interactive visualizations and applies to faceted search, cross-filtering, zooming, and panning. SMOKE pushes this down by partitioning the rid arrays (standalone, or part of rid indexes) by the predicate attribute. For the example above, SMOKE would partition the rid arrays in the backward index for `orders` by `shipdate`, so that C only reads the rid partition matching the parameter `:p1`. This technique applies to categorical as well as discretized continuous attributes. This makes it attractive for interactive visualizations since outputs are ultimately discretized at pixel granularity [JJHM14].

### 5.4.3 Grouping and Aggregation push-down

Interactions, such as cross-filtering [cro15], let users select marks in one view, trace those marks to the input records that generated them, and recompute the aggregation queries in other views based on the selected subset of input records. This pattern is precisely an aggregation query over the backward provenance of the user's selection. SMOKE pushes the group-by aggregation into provenance capture by partitioning the rid arrays on the group-by attributes, and incrementally computing the intermediate aggregation state. This works if the base and provenance consuming query primarily differ in terms of added grouping attributes, and effectively generates data cubes to answer the linage consuming aggregation queries. In contrast to building data cubes offline, which requires separate scans of the database, this optimization piggy-backs on top of the base query's table scans. As with prior work [GCB$^+$97; LJH13; HMT11], this optimization supports algebraic and distributive functions (e.g., SUM, COUNT, and AVG). To illustrate its importance, we evaluate it extensively in synthetic (Section 5.6) and real-world settings (Section 8.6).

## 5.5 Provenance semantics

In Section 4.6, we discussed how SMOKE can express and evaluate provenance semantics only after it has created its version of the provenance graph. However, provenance consuming applications may have a fixed logic that only requires a specific provenance semantics to operate on. In this section, we show how SMOKE can also push the evaluation of which- and how-provenance semantics into provenance capture. (Why- and where-provenance can be derived from either fine-grained provenance, which-provenance, or how-provenance, as we discussed in Chapter 4, and we omit further discussion.)

### 5.5.1 Which-provenance

SMOKE operates under which-provenance semantics for SPJAU queries using two simple techniques: SMOKE-W-I and SMOKE-W-D.

**SMOKE-W-I.** Under the former, instead of appending an rid in the provenance rid arrays or rid indexes, we first check if the rid is already present. If so, we do not append the rid; otherwise the rid is appended in the provenance rid arrays or indexes.

**SMOKE-W-D.** Under the latter, we first perform provenance capture as in Chapter 3. Then, to derive which-provenance, we remove duplicate rids from within the rid arrays. Note that this technique is similar to the ad-hoc technique that we introduce for ad-hoc evaluation of which-provenance queries in Section 4.6.1. However, there are two main differences between SMOKE-W-D and the ad-hoc evaluation of which-provenance queries of Section 4.6.1. First, note that SMOKE-W-D is essentially the pre-evaluation (and corresponding materialization) of every ad-hoc which-provenance query. Second, SMOKE-W-D happens before the issuance of which-provenance queries, whereas the ad-hoc evaluation of which-provenance queries happens at the moment of issuance of which-provenance queries.

Finally, note that both approaches can operate under either the INJECT or DEFER semantics of our transformational provenance capture that we introduced in Chapter 3. Also, recall that which-provenance is different than the fine-grained provenance of Chapter 3 only for specific types of queries, as we discussed in Section 4.6.1. To ease our discussion in our experimental section, we will focus only on INJECT semantics for both approaches (given that INJECT approaches have less overhead than their equivalent DEFER ones) and on queries for which which-provenance is different than the fine-grained provenance of Chapter 3.

### 5.5.2 How-Provenance

As we discussed in Section 4.6.3, how-provenance is grounded on the fact that relational operators can either combine tuples together or merge them. Based on this combining and merging, output tuples are associated with provenance polynomials that can both explain how an output tuple came into being as well used for the computation of semirings.

In Section 4.6.3, however, we considered the case where SMOKE perform provenance capture for transformational provenance semantics to only allow applications to compute provenance polynomials and semirings retrospectively. In this section, we discuss in more detail how SMOKE can derive provenance polynomials during the execution of the base query so that outputs are directly annotated with semiring-based values.

In this direction, we first need to partition relational operators into the ones that combine tuples together and operators that merge tuples together. Here we focus on relational operators for which how-provenance is well-defined (i.e., positive relational algebra extended with group-by aggregations). In this class, operators that combine tuples together include cross-products, natural joins, and $\vartheta$-joins. Operators that merge tuples together include group-by aggregations, set union, and set projection. In SMOKE's terms, operators that combine tuples together are binary (i.e., with two input sources) and have 1-to-1 backward semantics whereas operators that merge tuples have 1-to-N backward provenance semantics. This leaves out unary operators that have 1-to-1 backward semantics with one input source (i.e., bag projection and selection) which are treated specially.

Based on this classification, SMOKE can derive provenance polynomials in the following way. First, whenever SMOKE evaluates an operator that combines tuples together it associates the rids of the tuples, or other base values, with the abstract product operator. In other terms, instead of storing rids in two backward rid arrays it stores the results of the abstract product operator in a single array. Furthermore, whenever SMOKE evaluates an operator that merges tuples together, it applies the abstract sum operator on the corresponding rids or potentially other base values. Finally, note that for unary operators with 1-to-1 backward provenance semantics the base value of each input tuple is simply passed through to the output tuple that it contributes to.

## 5.6 Experiments

Our experiments in this chapter seek to show the performance of SMOKE on evaluating analytical provenance queries with and without our optimizations. To this end, and following our experiments in Chapter 3, we use the TPC-H database schema and focus our experiments around a subset of TPC-H queries (i.e., Q1, Q3, Q10, and Q12).

Next, we structure our experiments per class of optimization: pruning (Section 5.6.1), optimizations on provenance consuming SQL queries (Section 5.6.1), and optimizations on provenance semantics (Section 5.6.3). Within each class, we compare the performance of evaluating provenance queries in SMOKE with and (if applicable) without our optimizations. When possible, we also compare our techniques with state-of-the-art alternative approaches. Settings per experiment are inlined in our discussion.

### 5.6.1 Provenance Pruning

We start off our experiments by evaluating our optimizations on pruning input relations (Section 5.6.1.1) and provenance directions (Section 5.6.1.2).

#### 5.6.1.1 Pruning input relations

We start our discussion on provenance pruning from pruning input relations. Figure 5.1 compares the latency of Q3 and Q10, which read three and four relations, respectively, under three sets of conditions: no provenance capture, provenance capture for all input relations (non-optimized SMOKE-I), and SMOKE-I-based provenance capture for each individual single input relation. Furthermore, we did not evaluate Q1 because it has a single input relation. Finally, our findings on Q12 are the same as the ones we present here over Q3 and Q10, and we omit them to avoid redundancy.

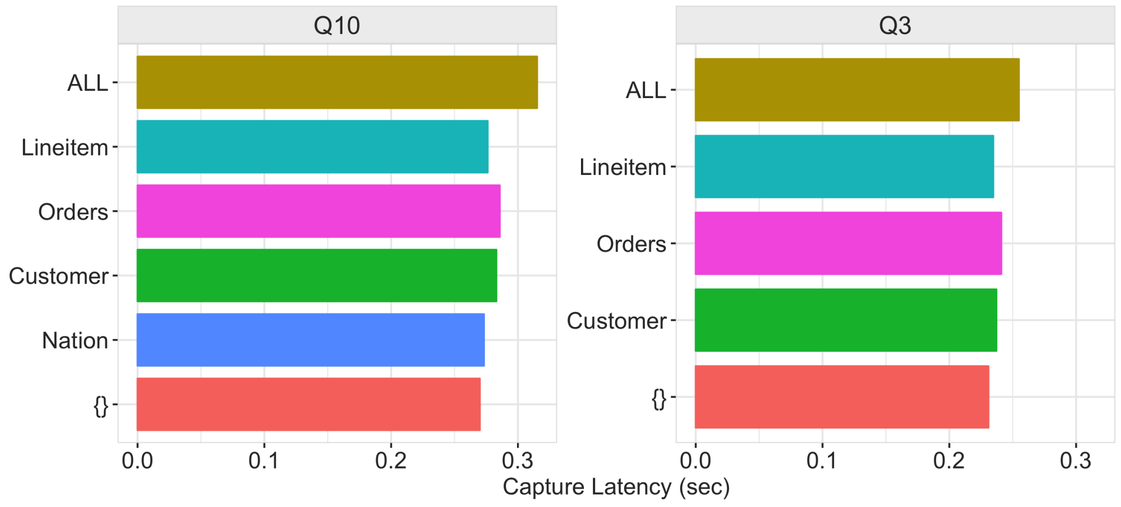


Figure 5.1: Provenance capture costs for different table pruning strategies. `ALL` refers to provenance capture for all tables. `{}` refers to not capturing provenance for any table. `Lineitem`, `Orders`, `Customer`, and `Nation` refer to capturing provenance only for the corresponding table and omitting provenance capture for all other tables.

As shown in Figure 5.1, pruning input relations from provenance capture reduces the overall provenance capture overhead. One interesting observation from our experiments is that the provenance capture cost for each individual relation is roughly the same. While this is expected, due to the nature of provenance capture (i.e., we need to write the same amount of rids for each table), there are two main differences worth pointing out.

First, `Lineitem` is the right-most table in the pk-fk joins of Q3 and Q10. This fact results in lower capture costs for `Lineitem` in comparison to the capture costs for other tables. This is due to the pk-fk optimization that we presented in Section 3.4.4. More specifically, recall that due to our optimization the forward and backward indexes for `Lineitem` are rid arrays. This is in contrast to the other tables for which we use rid indexes for both backward and forward provenance. As a result, capturing provenance for `Lineitem` has less provenance capture costs than for the other tables because rid arrays do not incur the initialization and reallocation costs of rid indexes.

Second, input relations that contribute multiple groups to the output are more likely to have higher capture costs than the ones with fewer groups. This is because we need to pay more initialization costs for the forward indexes. As a simple example, assume that we have 2 input tuples from the `Customer` table and 10 input tuples from `Orders` table (e.g., each customer has made 5 orders). Recall that SMOKE initializes rid arrays within rid indexes with an initial size of 10. This means that for the `Customer` table we would have to allocate 2 arrays (i.e., 20 bytes). In contrast, for the `Orders` table we would have to initialize 10 arrays (i.e., 100 bytes). Hence, although in total we have to write the same number of rids (i.e., number of outputs) in both forward indexes, the initialization costs result in different provenance capture costs. This observation is reflected in our experiments in Figure 5.1. For Q10, `Orders` has higher capture costs than `Customer` and `Customer` has higher capture cost than `Nation`. Similarly for Q3, `Orders` has higher cost than `Customer`.

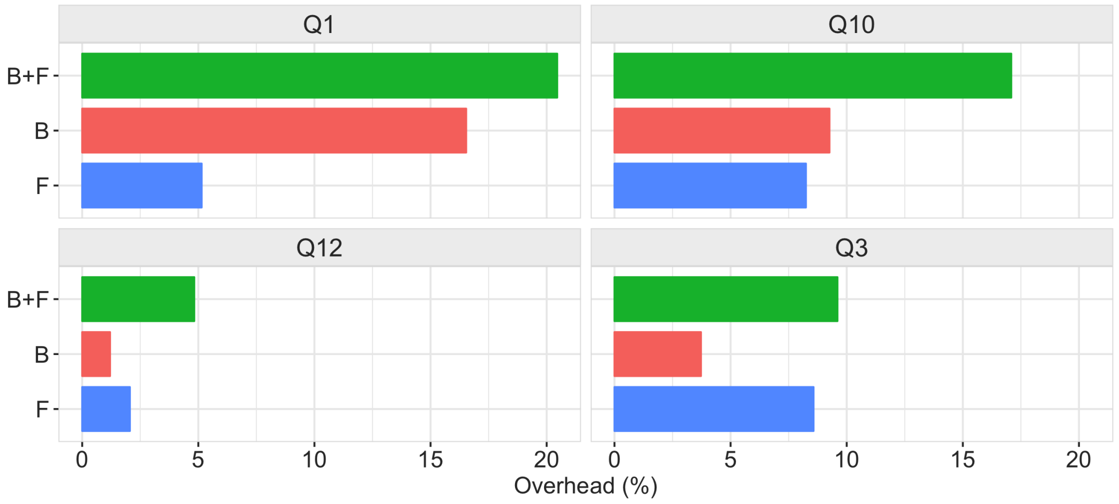


Figure 5.2: Provenance capture overhead on Q1, Q3, Q10, and Q12 for different provenance direction pruning strategies. `B+F` refers to capturing both backward and forward provenance for all input tables (i.e., no pruning). `B` refers to capturing only backward provenance for all input tables (i.e., omitting forward provenance for every input table). Conversely, `F` refers to capturing only forward provenance (i.e., omitting backward provenance for every input table).

#### 5.6.1.2 Pruning provenance directions

We conclude our discussion over provenance pruning by evaluating our pruning techniques for provenance directions. Figure 5.2 shows the overheads of tracking provenance on Q1, Q3, Q10, and Q12 using the following three strategies: capturing 1) both backward and forward provenance (i.e., `B+F`), 2) backward provenance but omitting forward provenance (i.e., `B`), and 3) forward provenance but omitting backward provenance (i.e., `F`).

As shown in Figure 5.2, pruning provenance directions reduces the capture overhead. The level of reduction, however, is subject to the type of the query and, consequently, the type of provenance indexes. For instance, Q1 is a group-by aggregation query. As a result, the backward provenance is stored in rid indexes which incur higher construction cost from the rid arrays that we use to store forward provenance. For Q12 and Q3, although the final operator to these queries is a group-by aggregation, the cost of capturing forward provenance is higher because forward provenance is stored in rid indexes. This is because the input to the group-by aggregation is a join for which forward provenance is captured in rid indexes. (Refer to our discussion in Section 5.6.1.1 for more details on the impact of forward indexes.) Note, however, that it is not always the case for such types of queries that the forward indexes will always have higher construction costs than constructing backward indexes. For instance, Q10 has the same query structure (i.e., join followed by group-by aggregation). Yet the backward indexes have higher construction costs than the forward ones, as is evident from our results in Figure 5.2.

Finally, note that for all experiments in Figure 5.2 we tracked provenance for every input table. In SMOKE, we can also track provenance for a subset of tables and only one provenance direction for each input table. Essentially, we can combine pruning of provenance directions with pruning of input tables.

***Takeaways:*** *Our experiments highlight that our provenance pruning optimizations can reduce the provenance capture overheads. The level of reduction is subject to which input relations and which provenance directions are pruned.*

### 5.6.2 Provenance Consuming SQL queries

We now present our experimental analysis on our push down optimizations for provenance consuming SQL queries. First, we present our results on the selection push down optimization (Section 5.6.2.1). Then, we present our results on data skipping and group-by aggregation push down optimizations (Section 5.6.2.2).

#### 5.6.2.1 Selection Push Down

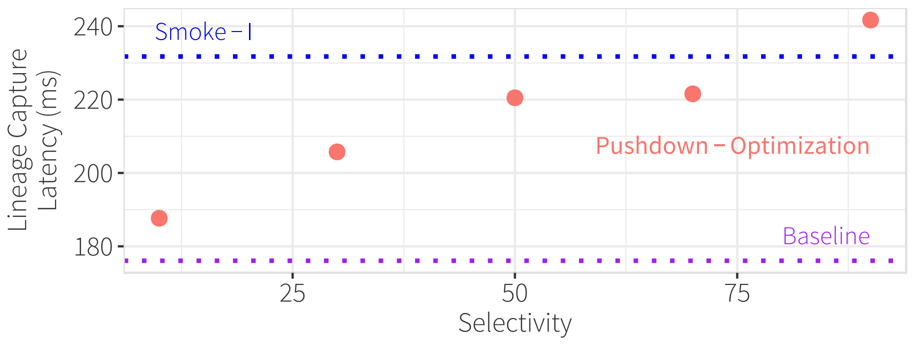


Figure 5.3: Provenance capture with selection push-down at varying selectivities of `l_tax < ?`. The crossover point between with and without push down is due to the additional cost of predicate evaluation before adding rids to the provenance indexes.

To evaluate the impact of the selection push down optimization, we used Q1 as the base query, and ran the following provenance consuming query:

```
SELECT * FROM L_B(Q1, Lineitem) WHERE l_tax < ?
```

Figure 5.3 plots the average and standard deviation base query latency when assigning `?` to 5 distinct `l_tax` values, along with the cost of SMOKE-I without selection push down, and LAZY. We find that the effectiveness of selection push down depends on the selectivity of the predicate. The overhead is linear with respect to the predicate selectivity, and there is a crossover point with SMOKE-I at high selectivities ($> 75\%$), where the overhead of evaluating the predicate for every input record outweighs the benefits of building a provenance index. We expect that increasing the predicate complexity (e.g., string comparisons, more predicate clauses) will likely shift the crossover point towards lower selectivities. These results suggest the value of cost-based methods to choose between the two.

#### 5.6.2.2 Data Skipping and Group-By Push Down

We explore the effectiveness of the data skipping and group-by push-down optimizations by incrementally building up an example motivated by the "Overview first, zoom and filter, details on demand" [Shn96] interaction paradigm. We focus only on zoom and filter because the base query generates the initial overview, while details on demand is the simple backward provenance query evaluated in Section 3.7.3.

We use TPC-H Q1 as the initial "Overview" base query (SF=2), and we render its output groups as a bar chart. There are four bars generated from 48%, 24%, 24%, and 0.06% of the `Lineitem` relation. Subsequent interactions (e.g., zoom in by drilling down and filter by adding predicates) will be expressed as provenance consuming queries that incrementally modify their preceding provenance consuming queries.

**No optimization.** Before considering optimizations, we first assess the effectiveness of provenance indexes on the evaluation of provenance consuming queries as compared to the lazy approach. Suppose users are interested in drilling into a particular bar to see its statistics grouped further by the month and year of the shipping date. This is expressed as a provenance consuming query $\mathrm{Q1_a}$ that changes $\mathrm{Q1}$ in two ways: (1) replaces the input relation with the backward provenance of the bar (i.e., $\mathrm{L_b}(\mathrm{o_a} \in \mathrm{Q1}(\texttt{Lineitem}),\texttt{Lineitem}))$ and (2) adds `Month,Year` of the shipping date to the `GROUP BY` clause.

We evaluate $\mathrm{Q1_a}$ for every value of $\mathrm{o_a}$ (not plotted). LAZY runs $\mathrm{Q1_a}$ as a table scan followed by filtering on Q1's group-by keys, grouping on year and month, and computing the same aggregates as Q1. SMOKE-I executes the same steps but evaluates $\mathrm{Q1_a}$ with secondary index scans as opposed to table scans. SMOKE-I performs best when the group cardinality is low ($0.06\%$ selectivity) and outperforms LAZY by 6.2×. For higher cardinality groups, SMOKE-I incurs the overheads of secondary index scans, as we also noted in Section 3.7.3. However, the performance of the two methods is similar because processing the high provenance cardinality (to compute the group-by aggregations in this case) dominates the execution of $\mathrm{Q1_a}$. A principled approach to avoid such high processing costs is using our workload-aware optimizations.

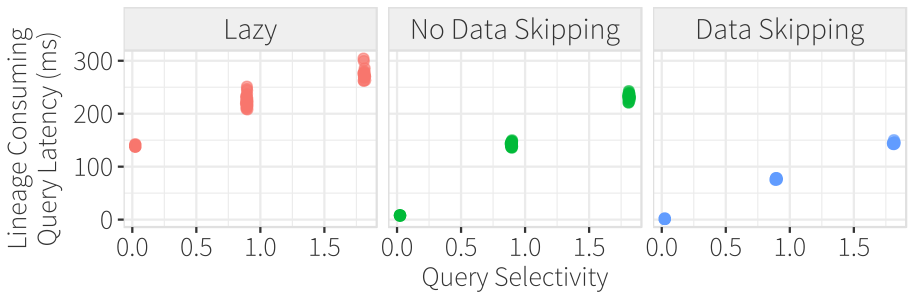


Figure 5.4: Provenance consuming query latency for different instrumentation approaches as the provenance consuming query's selectivity varies. Lazy requires table scans, No Data Skipping performs more efficient secondary index scans, and Data Skipping is $\leq$ 150ms because it only scans the relevant partition of the provenance index.

**Data skipping.** Suppose we know that the users want to filter the result of $\mathrm{Q1}$ (e.g., based on interactive filter widgets). Then we can push this logic into provenance capture using the data skipping optimization. We evaluate $\mathrm{Q1_b}$, which extends $\mathrm{Q1_a}$ with two parameterized predicates: `l_shipmode = :p1 AND l_shipinstruct = :p2`. Q1 is the base query for $\mathrm{Q1_b}$. To exercise push-down overheads, both are text attributes and thus more expensive to evaluate than numeric attributes. The provenance capture overhead was $0.22\times$ for SMOKE-I and $1.65\times$ with the data skipping optimization due to the additional cost of partitioning the rid arrays on the text attributes, but still lower than logical approaches (Figure 3.22).

Figure 5.4 plots the provenance consuming query latency for the selectivities of every possible combination of the predicate parameters. The LAZY baseline executes the provenance consuming query as a filter-groupby query over a table scan of `Lineitem`. Although provenance indexes substantially reduce query latency (No Data Skipping in Figure 5.4)—particularly for low predicate selectivities—it is bottlenecked by the secondary scan costs of backward provenance for high cardinality groups. In contrast, Data Skipping reduces even high selectivity queries by at least $2\times$ compared to LAZY, and is consistently below the interactive $150\text{ms}$ threshold [LH14]. This is because rid arrays are partitioned by `l_shipmode, l_shipinstruct` and the provenance consuming query is evaluated using indexed scans with only the rids needed to correctly answer the query.

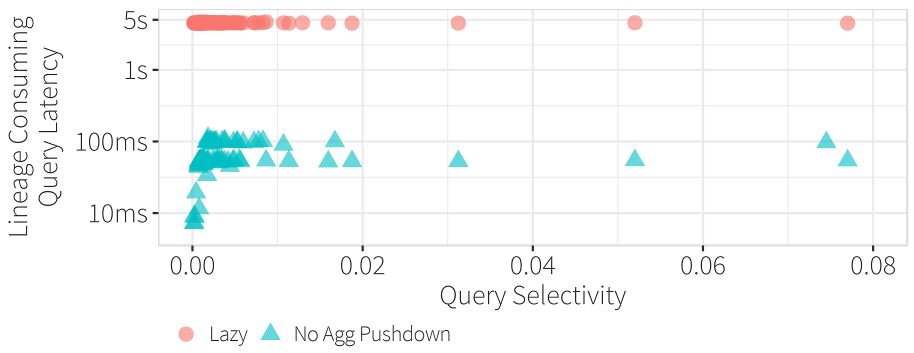


Figure 5.5: SMOKE-I reduces the provenance consuming query latency by $72.9\times$ on average as compared to LAZY. With aggregation push-down, the latency is $\approx 0$ms and we do not plot it.

**Group-by aggregation push-down.** After users filter and identify interesting statistics from the filter interactions in $Q1_b$, they may want to drill down further. If we know this upfront, SMOKE can pre-compute aggregates for new dimensions with the group-by aggregation push-down optimization. To evaluate this optimization, we compare LAZY against SMOKE-I (with and without the optimization) on $Q1_c$. $Q1_c$ changes $Q1_b$ by adding `l_tax` to the `GROUP BY` clause and setting the input relation to $L_b(o_c \in Q1_b(...)$,`Lineitem`$)$. For this experiment, we consider $Q1_b$ as the base query of $Q1_c$.

Figure 5.5 compares the provenance query latency under LAZY (red dots) against SMOKE-I without the optimization (blue triangles). The push-down optimization is not plotted because it takes $\approx 0$ms (i.e., just fetches the materialized aggregates). For completeness we vary the parameters of the backward provenance statement $L_b()$ for $Q1_c$ ($L_b(o_c \in Q1_a, ...)$) as well as for the base query $Q1_a$ ($L_b(o_a \in Q1, ...)$) of $Q1_b$ and report the provenance consuming query's latency for all combinations. Overall, LAZY takes $> 4$ seconds per $Q1_c$ instance while SMOKE-I takes from 7ms to $100$ms without the optimization and $\approx 0$ms with the optimization for all $Q1_c$ instances.

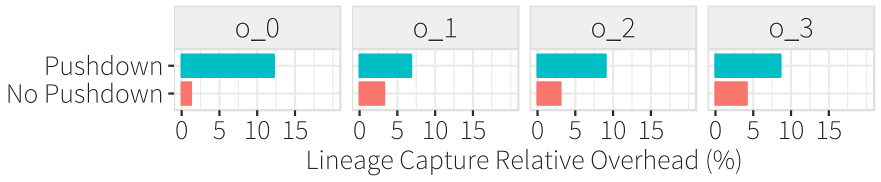


Figure 5.6: The average relative instrumentation overhead increases from $2.9\%$ without to $9.15\%$ with aggregation push-down.

Pre-computing aggregation statistics is not free, however. Figure 5.6 plots the provenance capture overhead for both SMOKE variants over to the non-instrumented lazy approach. We report the result for all 4 parameters to the base query $\mathrm{Q1_a}$'s backward provenance statement ($\mathrm{L_b}(\mathrm{o_a} \in \mathrm{Q1}, ...)$). The overhead of SMOKE-I is low compared to the overall cost of partitioning the rid arrays on `l_tax` and computing aggregates.

***Takeaways:*** *Our experiments highlight that provenance indexes are sufficient whenever the provenance cardinality is low for the complexity of future provenance consuming SQL queries. For higher provenance cardinalities, our workload-aware optimizations provide a principled way to push-down computation into provenance capture and optimize future provenance consuming queries. They also highlight tradeoffs that future optimizers for provenance-enabled database systems would need to consider.*

### 5.6.3 Provenance Semantics

We conclude our experiments by showcasing the performance of our ad-hoc and workload-aware techniques for evaluating and capturing which-provenance (Section 5.6.3.1) and how-provenance (Section 5.6.3.2).

#### 5.6.3.1 Which-Provenance

We evaluate the impact of our techniques on which-provenance capture using Q3. (Similar are our results on Q10 and Q12. For Q1, which-provenance and fine-grained provenance of Chapter 3 are the same).

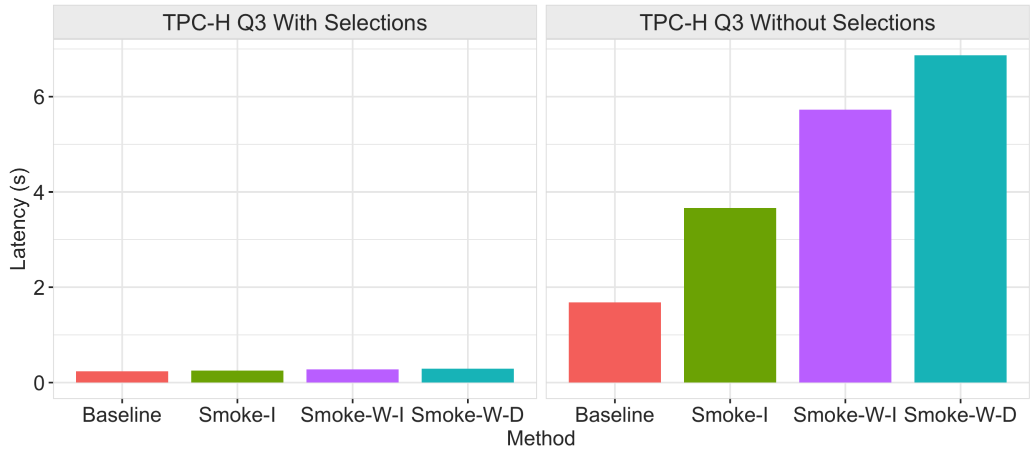


Figure 5.7: Capturing which provenance with inject (SMOKE-W-I) and defer (SMOKE-W-D) approaches in comparison to capturing transformational provenance with SMOKE-I and not capturing provenance at all BASELINE on TPC-H Q3 (with and without selections).

Figure 5.7 shows the performance of SMOKE-W-I and SMOKE-W-D (i.e., our workload-aware optimizations for materializing which-provenance at the time of the base query execution) in comparison to no provenance capture (i.e., BASELINE) and fine-grained provenance capture with INJECT semantics (i.e., SMOKE-I) on Q3 with and without selections. For Q3 with selections, the overheads of SMOKE-I, SMOKE-W-I, and SMOKE-W-D over the BASELINE are $.09\times$, $.15\times$, $.24\times$, respectively. For Q3 without selections, the overheads of SMOKE-I, SMOKE-W-I, and SMOKE-W-D over the baseline are $1.2\times$, $2.4\times$, and $3\times$, respectively. We make the following main observations over our results in Figure 5.7:

**Comparison of SMOKE-W-I with SMOKE-W-D.** Similar to the observations we made over INJECT- and DEFER-based capture techniques for fine-grained provenance (or lineage) capture in Section 3.7.1, SMOKE-W-I outperforms SMOKE-W-D because of the extra costs of deferring, in this case, parts of which-provenance capture. Recall that SMOKE-W-I de-duplicates rids at the moment we are appending them in rid indexes. In contrast, SMOKE-W-D first appends duplicate rids in rid indexes to only after de-duplicate them. Hence, SMOKE-W-D has to pay the extra cost of storing redundant rids. However, note that storing duplicate rids (i.e., the first step of SMOKE-W-D) is essentially equivalent to the SMOKE-I approach which has less overhead than SMOKE-W-I. As such, by deferring the de-duplication, SMOKE-W-D can be used by applications that want to avoid blocking on which-provenance capture on base query execution using SMOKE-W-I (for the time $\text{latency}_{\text{SMOKE-W-I}} - \text{latency}_{\text{SMOKE-I}}$).

**Comparison with SMOKE-I.** As shown in Figure 5.7, SMOKE-W-I and SMOKE-W-D always incur overheads higher than SMOKE-I. This is because SMOKE-W-I and SMOKE-W-D have to perform the extra step of de-duplication that SMOKE-I does not perform.

**Impact of selections.** Similar to our results on fine-grained provenance capture (Section 3.7.2), selections also play an important role on the overhead of our which-provenance techniques. For instance, the overhead over the BASELINE of SMOKE-W-I on Q3 with selections is $0.15\times$ whereas without selections is $2.4\times$. Similarly, the overhead of SMOKE-W-D increases from $.15\times$ for Q3 with selections to $3\times$ for Q3 without selections.

**Ad-hoc evaluation.** Finally, recall from our discussion in Section 5.5 that SMOKE-W-D corresponds to the ad-hoc evaluation of all possible which-provenance queries. (The ad-hoc evaluation of which-provenance queries is presented in Section 4.6.1.) As such, the performance of ad-hoc evaluation of why-provenance queries (not shown in Figure 5.7) is as follows: At the moment of provenance capture, we need to capture fine-grained provenance provenance. This is the performance of SMOKE-I in Figure 5.7. Then, the latency of each which-provenance query corresponds to a portion of $\text{latency}_{\text{SMOKE-W-D}} - \text{latency}_{\text{SMOKE-I}}$ for rid de-duplication. The worst de-duplication performance, and hence the worst case for which-provenance queries, comes from cases where we need to de-duplicate large rid arrays.

#### 5.6.3.2 How-Provenance

So far we have discussed how to capture and evaluate which-provenance semantics using both our ad-hoc and workload-aware techniques. Here, we further evaluate SMOKE on capturing and evaluating how-provenance semantics by focusing on two semirings: weight/cost and derivability semirings [KIT10].

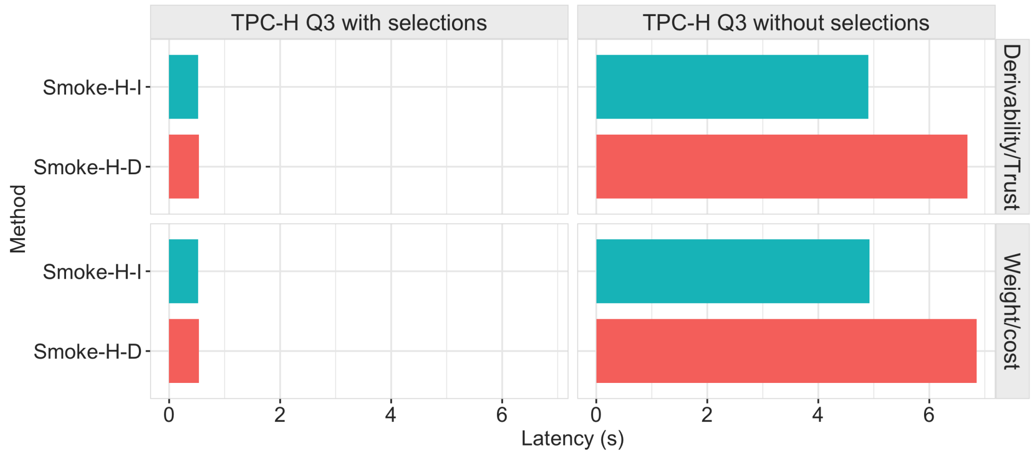


Figure 5.8: Latency of SMOKE-H-I and SMOKE-H-D on TPC-H Q3 with (left) and without (right) selections for capturing weight/cost (up) and derivability/trust (down) semirings. For comparison purposes, the latency of BASELINE and SMOKE-I are depicted in Figure 5.7 and we omit them here to avoid redundancy.

Figure 5.8 shows the performance of SMOKE for capturing how-provenance under the weight/cost and derivability/trust semirings. Following the notation for which-provenance, SMOKE-H-I refers to materializing the result of semirings during base query execution while SMOKE-H-D refers to materializing the result of semirings after we have first captured fine-grained provenance on the base query (i.e., by evaluating how-provenance queries in the ad-hoc way that we showed in Section 4.6.3 for every possible output of the base query).

Our observations over the results of Figure 5.8 follow the ones we made for which-provenance. This is expected as which-provenance is also a semiring (often referred to as lineage semiring [KIT10]). To this end, we omit further discussion and we refer readers to our observations on the which-provenance results. One interesting note, however, is that the actual latency results that one should expect during the evaluation of a semiring depends on the complexity of the semiring. For instance, the latency of SMOKE-H-D for weight/cost and derivability/trust in Figure 5.8 are ~6.8s and ~6.6s, respectively (i.e., a difference of ~200ms). While the semirings in our experiments belong in equivalent time complexity classes, we note that the complexity of the abstract sum and product operations of semirings can result in different capture latencies.

***Takeaways:*** *Our experiments highlight that our workload-aware optimizations can significantly improve the overall latency required for the evaluation of ad-hoc provenance queries. However, the ad-hoc evaluation of each individual provenance query for different provenance semantics can be fast enough due to the underlying captured fine-grained provenance graph. Finally, the extent to how fast we can evaluate provenance semantics in an ad-hoc way or materialize provenance semantics through workload-aware optimizations depends on the complexity of the provenance semantics.*

## 5.7 Conclusions

In this chapter, we presented our workload-aware optimizations on the evaluation of provenance analytics and compared them with our techniques for ad-hoc evaluation. Overall, our results show evidence that in many cases our ad-hoc evaluation techniques may be sufficient given the end-latency objective of provenance applications. Furthermore, we showed evidence that for data-intensive provenance applications that require several orders of magnitude improvements over the ad-hoc evaluation (e.g., interactive visualization) our optimizations (e.g., data skipping and group-by aggregation push down) provide principled ways for provenance query latency reduction. Finally, we showed that provenance capture overheads may increase or decrease depending on the optimization.

# Chapter 6

# Physical Plan Instrumentation

In the previous chapters, we showed how we can change physical plans to piggyback the provenance capture logic within the query execution logic (Chapter 3), how we can query the captured provenance information to perform analytics (Chapter 4), and how we can optimize provenance analytics by pushing them into the capture phase if analytical provenance workloads are known at the moment of provenance capture (Chapter 5).

Now, consider implementing the provenance capture techniques and optimizations within a database. At minimum, we would have to change the source code of the whole physical algebra, possibly duplicating it to account for cases when we want to perform only normal query execution, perform INJECT provenance capture, perform DEFER provenance capture, and introduce our push-based workload-aware optimizations.

In fact, the first version of SMOKE was following a similar design: a database engine with operators implementing both their logic and the provenance logic. The provenance logic was activated through switches passed as directives to the optimizer which was responsible for initializing physical operators with the switches on. While this was somewhat fine for provenance capture given the minimal changes required, adding the workload-aware optimizations ended up extending the logic of each physical operator. The overall result was going from physical operators implementing their standard logic to having physical operators augmented with external logic—dwarfing the standard logic and polluting the overall physical algebra that SMOKE supports with an ever-increasing injected code.

Stepping up a level, however, even if we did so, the end-product is a custom database with provenance capture and query capabilities which, while important due to the large number of application domains that provenance supports, does not expose any extensibility mechanisms to introduce other modules that operate similarly to the provenance manager.

To this end, this chapter introduces the current version of SMOKE that exposes principled mechanisms for physical plan instrumentation to facilitate the development of applications that operate on how queries are executed, similarly to how provenance managers operate.

## 6.1 Introduction

As we noted in Chapter 1, an ever increasing number of user-facing application domains (e.g., data visualizations; data profiling; data explanation; and data debugging) and in-database modules (e.g., provenance managers; online query optimizers; online and adaptive database designers; or self-regulating managers) rely their logic on how queries are executed. Unfortunately, extending a database engine to support the development of techniques in these domains, while important, remains challenging.

The two predominant approaches to introduce such techniques is either by rewriting databases or working around the SQL interface both of which come with shortcomings. For instance, in the previous chapters, we showed how we rewrote the physical algebra of SMOKE to perform query execution and provenance capture at the same time—which is an instance of the former approach—as well as logical provenance capture approaches that perform provenance capture by working around the SQL interfaces. The former approach, while performant for the task of provenance capture, results in a database that provides no mechanisms for other techniques that operate on how queries are executed. The latter approach essentially treats SQL as a mechanism for provenance capture and comes with significant performance penalties, rendering such provenance systems inapplicable for data-intensive tasks, as we demonstrated experimentally in Chapters 3 and 5.

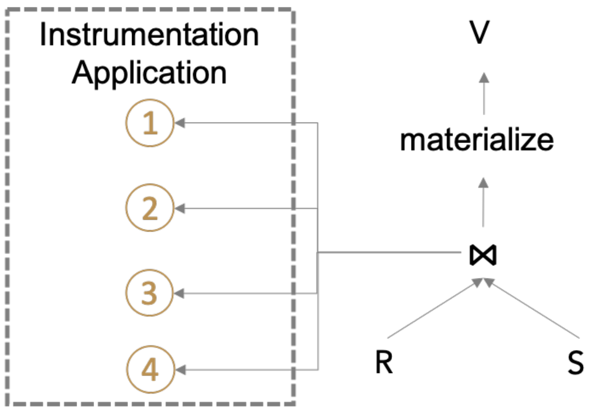
To address the problem of expressing and optimizing techniques that operate on how queries are executed, without having to rewrite databases and finding workarounds, our main idea is to enable *physical plan instrumentation* (i.e., allow third-party applications to operate on physical plans), as we also noted in Chapter 2. A naive approach to follow in this direction is to simply send the physical plan as generated by the query optimizer to applications to apply their logic on. In fact, more recent designs of database systems already provide this functionality [Pos13]. Manipulating physical plans, however, comes with many challenges, as we will see in this chapter, that external applications have to deal with on their own. In fact, some of these challenges involve operations that require changes of the underlying database that applications have no control on by just operating on physical plans.

To this end, in this chapter, we introduce the underlying mechanisms for physical plan instrumentation that SMOKE exposes to instrumentation applications as well as the underlying changes that we made to SMOKE in support of such mechanisms.

Our mechanisms aim to introduce a core set of instrumentation capabilities that are of common use across domains, yet involve multiple technical challenges that instrumentation applications would have to otherwise address on their own. Next, we discuss several desiderata by instrumentation applications that our mechanisms aim to provide.

## Instrumentation Points

Across domains, instrumentation applications need to embed their logic within the query execution logic. To illustrate, consider a query `V=SELECT * FROM R,S WHERE P.pid=S.pid` and assume that SMOKE evaluates `V` with a nested loop (NL) join; Figure 6.1 shows the plan (middle) and source code (right). Furthermore, the source code in Figure 6.1(right) is annotated with circled numbers ①, ②, ③, and ④ corresponding to some points in the logic of the NL join. Such points could be used by instrumentation applications to integrate their logic within the query execution. Let us consider four simple



```
Output V;
for(r in R){
 for(s in S){
  if(r.pid == s.pid){
    ①
    V.append(merge(r,s))
    ②
  }else{
    ③
  }
 }
 ④
}
```

Figure 6.1: Physical plan (middle) and source code (right) for our example query `V=SELECT * FROM R,S WHERE P.pid=S.pid`. Circled numbers (i.e., ①, ②, ③, and ④) in the source code denote some points in the logic of the nested loop join that instrumentation applications (left) could use to integrate their logic.

instrumentation applications including monitoring, online optimizer, negative provenance manager, and positive provenance manager:

**Monitoring.** If a monitoring application wants to measure how much time is spent on materializing join results, the implementation for ① and ② could be `time_start = NOW()` and `total_time += NOW() - time_start`, respectively, with `NOW()` denoting the current time. Such monitoring results could be used in a number of ways ranging from self-regulating components to profiling dashboards.

**Online Optimizer.** Similarly, if online optimizers, such as adaptive join ones [SQL18; Ora17], want to get online the join cardinality, the implementation of ① could be as simple as `join_cardinality++`. Based on this knowledge, such optimizers could decide to change the nested loop join to hash join, among other operations, as we will see.

**Negative Provenance Manager.** Furthermore, if a negative provenance manager wants to materialize tuples that did not satisfy the join it could use the points ③ and ④. In ③ we could materialize the S tuples that did not join with a given R tuple, while in ④ we could materialize R tuples that did not join with any S tuple if in ① or ② we keep track if there was a match. Based on these results, negative provenance managers could provide their common functionalities including data debugging or data profiling to name a few.

**Positive Provenance Manager.** Finally, if a positive provenance manager wants to perform positive provenance capture, it can use ② which is precisely the point where we know what input records contributed to what output records. This also illustrates the main difference from our initial approach with provenance capture on nested loop joins, that we showed in Section 3.4.10. If a database was providing such points where we could integrate our provenance logic in a principled manner within physical operators, we would not have to hard code changes to the physical algebra, as we did with the first version of SMOKE.

Exposing instrumentation points is the most important and technically challenging requirement of instrumentation applications—and hence the main focus of this chapter. Besides instrumentation points, however, instrumentation applications are in need of several other mechanisms to facilitate the implementation of their logic that we outline next.

## Actions on Plans

Instrumentation applications may need to change physical plans by means of modifying, replacing, removing, and adding operators in the plan either during or before query execution. To illustrate, consider our NL join example again. The choice of the optimizer for an NL join physical operator may be poor due to erroneous estimation of statistics involved in cost-based decisions (e.g., estimated join cardinality). Online query optimization techniques, such adaptive joins, update statistics at runtime, such as the `join_cardinality` that we discussed above, and based on them may change the plan to a more efficient hash-based join. Conversely, hash-based joins may change to nested loop joins, say, because during execution there was a change on the memory budget available rendering maintaining a hash-table expensive. Similarly, applications such as Smooth Scan [BGIA+18] may want to change selection scans to indexed scans whereas applications such as probabilistic predicates [LCKC18] may want to change predicates of a selection, add selections in a query, or even remove selections from a query.

### Operator State Access

Moreover, instrumentation applications may need to read and write the state maintained by physical operators. For instance, consider again our example query this time implemented as a hash-based join. In Section 3.4.4, we showed how we can augment hash tables maintained by hash-based joins with rid arrays to optimize the provenance capture on hash-based joins. Similarly, online query optimization techniques in the presence of events such as exceeding memory may want to read hash tables or intermediate relations to perform compression. Finally, actions on plans that we presented above may also need to be followed by accessing the state of operators. For instance, if in our example we change the projection clause to include some of the attributes of `R`, then the hash table maintained by the hash-based join may need to change accordingly.

### Access to Storage

Additionally, besides only reading and writing the state of physical operators, instrumentation applications may need to a) create and maintain their own storage either to implement their instrumentation logic or to create a state that will be used post-instrumentation and b) read and write the storage maintained by the database. For instance, in our discussion on instrumentation points on monitoring, `time_start` is a variable used to implement the instrumentation logic while `total_time` is a variable that will be used post-instrumentation by clients of the monitoring module. Furthermore, adaptive physical database designers, such as database cracking, need access to the internal storage to reorganize it during query execution. Similarly, our positive provenance manager, as introduced in Chapters 3 to 5, needs access to storage to materialize physical representations of provenance that it can later use to answer analytical provenance queries.

### Scheduling

Also, instrumentation applications may need to schedule their instrumentation logic relative to the execution of individual operators, pipelines, or whole physical plans. For instance, in our example of provenance tracking on our toy example, paying the whole cost of provenance tracking during query execution by embedding the tracking logic in 2 may result in overhead that some applications (e.g., interactive visualizations) may not tolerate. In this direction, instrumentation applications need both automated and manual ways to either defer their whole logic after execution or partially inject some logic and defer the rest.

### Notifications

Finally, instrumentation applications may also need to specify runtime events, get notified when these arise, and act upon them in an application-specific way. For instance, the decision on what to defer and what to inject for provenance capture may be either hard-coded pre-execution but, in the general case, it is driven by events raised during query execution (e.g., the CPU cycles spent on tracking provenance). To enable such functionality, instrumentation applications should be able to specify events in the form of conditions (e.g., CPU cycles spent on an operation is above some threshold), take the control when conditions are met, and perform actions based on them—which is a typical design for reactive systems.

To account for the desiderata discussed above, SMOKE introduces a physical plan instrumentation framework with several components (i.e., Points and Instrumentors, Actions, Scheduler, Storage Manager, and Announcer) exposing the desired mechanisms (i.e., instrumentation points, actions on plans, operator state access and storage access, scheduling, and notifications, respectively). Throughout this chapter, we outline and address the technical challenges behind each component as well as present our techniques for changing components of SMOKE (i.e., compiler and physical algebra) in support of such mechanisms.

## Contributions and Chapter Roadmap

In the rest of this chapter, we start by introducing the architecture of SMOKE extended with the physical plan instrumentation framework as well as examples of instrumentation applications within SMOKE (Chapter 2). Then, we present our contributions as follows:

- We introduce a specification of instrumentation points (Points) on individual physical operators (i.e., selection, hash-based group-by aggregations, hash-based joins, nested-loop joins, cross products, projection, and materialization) as well as on pipelines and plans. Furthermore, we associate each point with data flows (i.e., streams of records or data structures) that applications can operate on at these points (Section 6.3).

- The specification of points and their associated data flows provide logical instrumentation semantics. For applications to actually use this semantics they need to implement programmatic interfaces, termed Instrumentors. Each instrumentor exposes instrumentation functions that correspond to instrumentation points and take as input the associated data flows. To account for flexibility in the implementation of the instrumentation logic, we introduce different types of instrumentors (i.e., interpretation- and compilation-based) each with unique properties. (Section 6.4).

- We introduce our Scheduler that allows applications to defer or inject instrumentation logic and impose execution orders of instrumentors. (Section 6.5).

- We introduce our Storage Manager that allows applications to create, read, and write their state within the storage of SMOKE; access the internal state of operators and pipelines (i.e., hash tables as well as intermediate and output relations); as well as access the internal storage of SMOKE. (Section 6.6)

- We introduce our Announcer component that allows applications to specify run time conditions as well as functions to-be-executed when conditions are met. (Section 6.7)

- We introduce our Actions component that allows applications to modify, replace, add, or remove physical operators. (Section 6.8)

- We introduce our instrumentation-aware compiler that, given an instrumented physical plan, generates the source code. Furthermore, we outline the changes in the physical algebra that we made in support of the instrumentation mechanisms. (Section 6.9).

Expressiveness- and performance-wise, we evaluate our framework throughout Part II. We finish this chapter with a discussion on potential concerns around physical plan instrumentation and conclusions based on instrumentation applications outlined in this chapter and the provenance techniques that we have already introduced in Chapters 3 to 5.

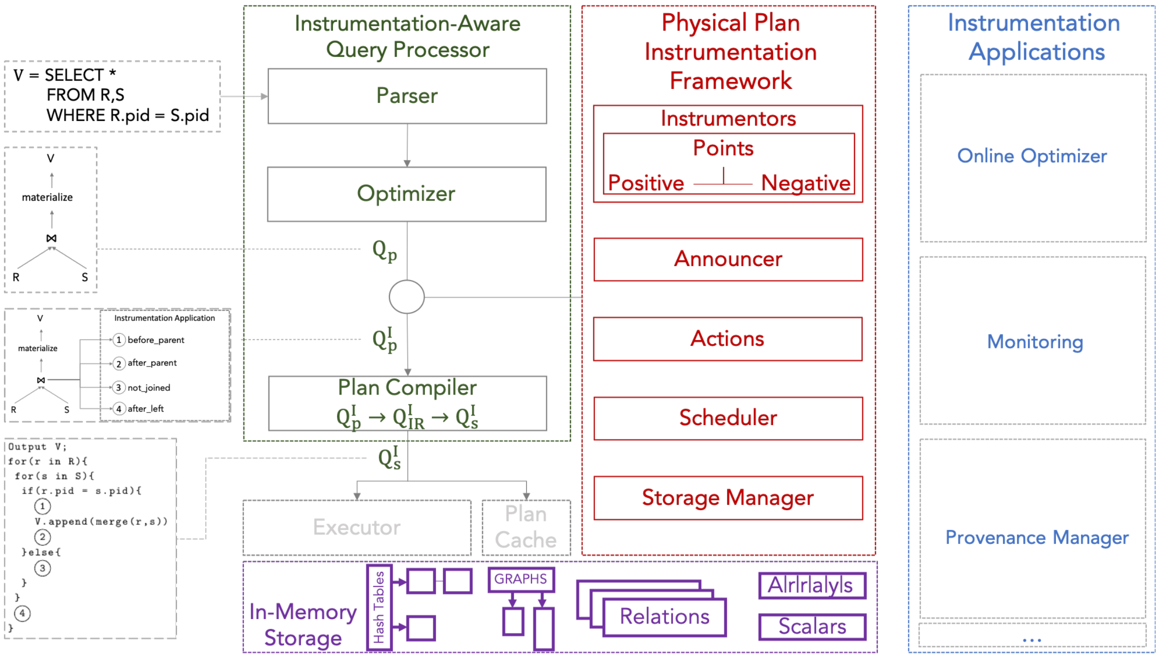
## 6.2 Architecture of SMOKE and Examples

Given a physical plan, the problem of focus in this chapter is to provide the necessary mechanisms to instrumentation applications to generate and execute an *instrumented physical plan* that extends, alters, or analyzes the initial physical plan. To this end, SMOKE provides a physical plan instrumentation framework and the necessary underlying database architecture to expose instrumentation APIs and management capabilities to instrumentation applications. Next, we first present an overview of the architecture of SMOKE for instrumenting physical plans (Section 6.2.1). Then, we present motivating examples of instrumentation applications to illustrate the instrumentation process (Section 6.2.2).

### 6.2.1 Architecture

The architectural design of SMOKE is composed out of three major components relevant to instrumenting physical plans (i.e., Instrumentation-Aware Query Processor, Physical Plan Instrumentation Framework, and In-Memory Storage) as depicted in Figure 6.2.

**Instrumentation-Aware Query Processor.** Given a query from a client, the query processor first parses and optimizes it by passing it through the Parser and Optimizer modules. The end result of the Optimizer is a physical plan. At this point, if there are no instrumentation applications that need to instrument a plan, the Compiler module compiles the physical plan



```
V = SELECT *
    FROM R,S
    WHERE R.pid = S.pid
```

```
Output V;
for(r in R){
 for(s in S){
  if(r.pid = s.pid){
   ①
   V.append(merge(r,s))
   ②
  }else{
   ③
  }
 }
 ④
}
```

Figure 6.2: Architecture of SMOKE.

to source code. It does so by first compiling the physical plan to its internal IR that it then compiles into source code. Note that the compilation from physical plan to the internal IR of SMOKE follows the well-known producer-consumer query compilation model [Neu11], as we discussed in Chapter 2. The end result is passed to the Executor which compiles the source code to binary, links it with SMOKE, and executes it as such to provide the results back to the client. If there are instrumentation applications that need to instrument the plan, then they are handed over the physical plan. Instrumentation applications instrument the plan using the Physical Plan Instrumentation Framework and return back to the Instrumentation-Aware Query Processor an *instrumented physical plan*. Then, the Compiler module performs the same operation as before (i.e., compilation to its internal IR and then to source code). This time, however, it compiles an instrumented physical plan which is different as a task from compiling a physical plan, as we will see in Section 6.9.

**Physical Plan Instrumentation Framework.** To allow instrumentation applications to instrument physical plans SMOKE exposes a Physical Plan Instrumentation Framework.

The framework consists of several modules each exposing a set of APIs and associated interfaces to allow applications to push their logic within physical plans (i.e., Points and Instrumentors); change, replace, remove, and add physical operators included in a physical plan (i.e., Actions); schedule their instrumentation logic relative to the query execution logic (i.e., Scheduler); get notified when events arise during query execution and react to such events in an application-specific way (i.e., Announcer); and read, write, and modify the physical database design of the database and the state of physical operators as well as create, read, write, and modify their internal storage (i.e., Storage Manager). Instrumentation applications can implement their logic using the Physical Plan Instrumentation Framework by implementing interfaces and using API functions exposed from each component of the framework, as we will see throughout this chapter.

**In-Memory Storage.** Finally, the underlying In-Memory Storage of SMOKE is composed out of scalars, arrays, relations, graphs, and hash tables that both physical plans and instrumented physical plans can create, read, write, and modify. Note that instrumentation applications also have access to storage maintained by physical plans so that they can access the state of operators. We provide more technical details on the storage of SMOKE and how applications can use it in Section 6.6 when we present the Storage Manager of the Physical Plan Instrumentation Framework.

Having described the main components relevant to physical plan instrumentation, we next discuss how SMOKE processes queries under no instrumentation to contrast it with how queries are processed under instrumentation.

**Query execution under *no* instrumentation.** Consider our toy join example and the architecture of SMOKE in Figure 6.2. During normal query execution, SMOKE takes as input $\mathrm{V}$, parses and optimizes it, to get a corresponding physical plan $\mathrm{Q_p}$. At this point assume that there are no instrumentation applications that need to instrument it. SMOKE then compiles the physical plan into source code and executes it as such.

**Query execution under instrumentation.** If there are applications that need to instrument the physical plan of $\mathrm{V}$, SMOKE's compilation and execution logic changes. Given our query

V, SMOKE will parse and optimize it to get a physical plan $Q_P$. (This phase is the same with the normal query execution.) At this point, SMOKE sends the physical plan to instrumentation applications that want to instrument $Q_P$. To illustrate, Figure 6.2(right) shows three applications (i.e., `Online Optimizer`, `Monitoring`, and `Provenance Manager`) that are handed over the physical plan. In turn, each instrumentation application instruments a physical plan by using functions and implementing interfaces of the Physical Plan Instrumentation Framework. (We describe how these example instrumentation applications instrument physical plans in Section 6.2.2.) The overall result of this phase is an instrumented physical plan $Q_P^I$ that is returned by the Physical Plan Instrumentation Framework when instrumentation applications have finished subscribing their instrumentation logic. Then, the Compiler of SMOKE's Instrumentation-Aware Query Processor takes as input the instrumented physical plan $Q_P^I$ and compiles it into source code $Q_S^I$. At runtime, the instrumented physical plan executes the logic of the instrumentation applications along with the logic of the initial physical plan (if this has not changed due to instrumentation).

To illustrate the instrumentation process, we next present motivating examples of instrumentation applications.

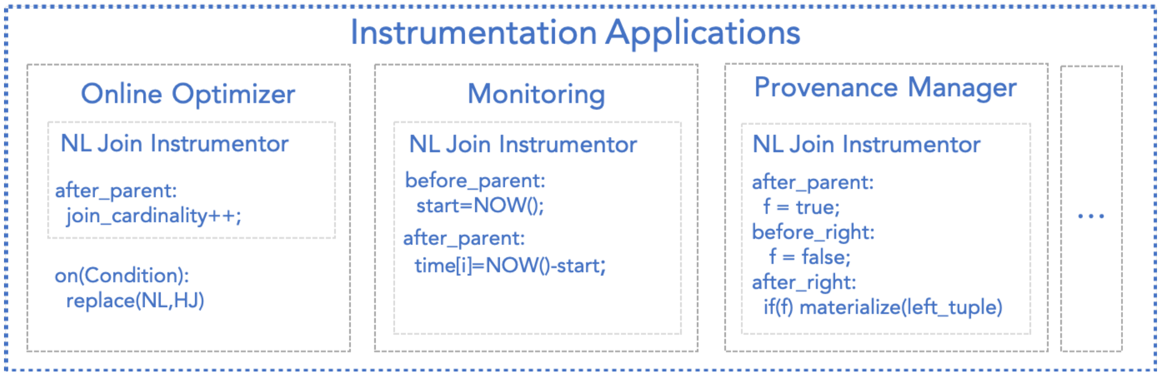


Figure 6.3: Example Instrumentation Applications.

### 6.2.2 Motivating Examples of Instrumentation Applications

Consider the instrumentation applications depicted in Figure 6.3 (i.e., `Online Optimizer`, `Monitoring`, and `Provenance Manager`). Next, we outline how they can be implemented within SMOKE. Whenever we use functionality provided by a component of the Physical Plan Instrumentation Framework, we also provide forward pointers to relevant sections that describe in more detail the functionality that we used.

**Example 5 (Online Optimizer)** The `Online Optimizer` in Figure 6.3 tracks the `join_cardinality` by implementing the `after_parent` instrumentation point of the NL join operator (Section 6.3). To do so, it needs to implement an *instrumentor* which is an interface associated with an NL join operator and exposes functions associated with its instrumentation points (Section 6.4). Furthermore, it implements an `on(Condition)` function which is part of the Announcer component (Section 6.7) and its purpose is to be called when the `Condition` is met. A typical `Condition` for our example could be that the join selectivity obtained at runtime using the tracked `join_cardinality` has surpassed a limit. If this condition is met, then the `Online Optimizer` uses the `replace` function to replace the nested loops join with an equivalent hash-based join implementation (Section 6.8). Finally, note that the `join_cardinality` and other variables used in the calculation of the `Condition` can be maintained by the storage of SMOKE (Section 6.6).

**Example 6 (Monitoring)** As another example, the `Monitoring` application tracks the time spent on materializing the results of the join. To do so, it can `register` an array `time[]` and a scalar `start` into the storage of SMOKE using the Storage Manager (Section 6.6). The array `time[]` maintains the time taken to materialize each join result, and the scalar `start` tracks the current time right before the materializer consumes a join result. Then, it implements the `before_parent` and `after_parent` functions (Section 6.3) of the NL join instrumentor (Section 6.4): `before_parent` assigns the current time to the scalar `start`, and `after_parent` computes the time taken on materializing a result (using the `start` time and the current time NOW()) and appends it to the array `time`.

**Example 7 (Provenance Manager)** Finally, the `Provenance Manager` tracks negative provenance from the left side (i.e., the records from the left side that did not contribute to the join result). To do so, it implements the `after_parent`, `before_right`, and `after_right` instrumentation points (Section 6.3), which are part of the NL join instrumentor (Section 6.4), with relevant logic. While not shown in Figure 6.3, the logic pushed on these instrumentation points can also be deferred by our Scheduler component (Section 6.5). Furthermore, note that the Scheduler component is also responsible for ordering the instrumentors. For instance, here all three applications push their logic within the `after_parent` point. Since there is no ambiguity between them, the Scheduler decides to execute them based on the order that they register on the `after_parent`, as we will see in Section 6.5.

In this section, we presented the architecture of SMOKE, described on a high-level the components of the Physical Plan Instrumentation Framework of SMOKE, and presented motivating examples of the overall instrumentation process. Next, we dive deeper into each component of the Physical Plan Instrumentation Framework to describe and address its challenges. We start our discussion from the instrumentation Points.

## 6.3 Instrumentation Points

To allow instrumentation applications to integrate their logic within the query execution logic the first step is to introduce the points where instrumentation logic can be integrated which is the focus of this section. Based on these points, we can define instrumentors which are the programmatic interfaces that applications can use to implement their instrumentation logic, as we will see in Section 6.4.

**Challenges.** There are two technical challenges behind introducing instrumentation points. The first challenge is to break down the logic of physical operators into meaningful fragments so that we can expose a complete and semantically meaningful set of points available for instrumentation. The second challenge is grounded on the fact that each point in the logic of a physical operator is associated with information that applications could operate on.

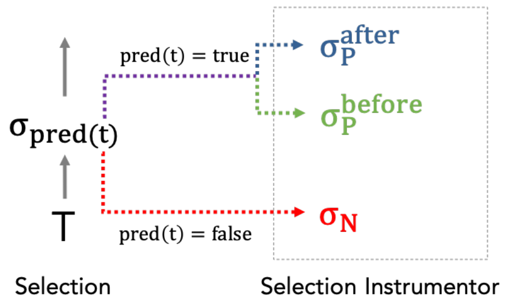
For instance, in the logic of a nested loop join the point following the satisfaction of the predicate is logically associated with the tuples that satisfy the join predicate that provenance managers could use for provenance capture purposes. Then, our second challenge is how to associate each instrumentation point with information that can be processed at these points.

**Solution Overview.** To address these challenges, in this section we break down the logic of each physical operator (in the physical algebra of SMOKE) into code fragments to which we assign instrumentation points. Then, we review each point and associate it with information that applications could process on these points. Overall, the main design principles behind the introduction of instrumentation points are three: 1) every semantically meaningful code fragment is associated with before and after instrumentation points (e.g., for monitoring applications that need to wrap the logic of the fragment with start and end clock ticks), 2) every code fragment associated with data flows (i.e., stream of records) or data structures (e.g., hash tables) due to query processing is also associated with instrumentation points so that instrumentation applications can operate on them (e.g., for provenance capture), and 3) for operators that filter data flows (e.g., the selection operator filters out records that did not satisfy the selection predicate) we introduce new code fragments so that applications can have access to filtered out data flows (e.g., for negative provenance capture purposes).

Next, we introduce the instrumentation points of selections, hash-based group-by aggregations, and (hash-based and nested loop) joins to illustrate the main concepts behind our approach. For completeness, the full set of instrumentation points that SMOKE supports in its physical algebra is in Table 6.1 at the very end of this section.

### 6.3.1 Selection

Consider the plan and corresponding source code for the selection operator under instrumentation in Figure 6.4. SMOKE's instrumentation framework exposes three instrumentation points on selection (i.e., $\sigma_{\mathrm{P}}^{\mathtt{before}}$, $\sigma_{\mathrm{P}}^{\mathtt{after}}$, and $\sigma_{\mathrm{N}}$). $\sigma_{\mathrm{P}}^{\mathtt{before}}$ corresponds to the code fragment that will be executed for records that satisfy the selection predicate but before the parent operator of the selection in the physical plan has consumed a record. Similarly,

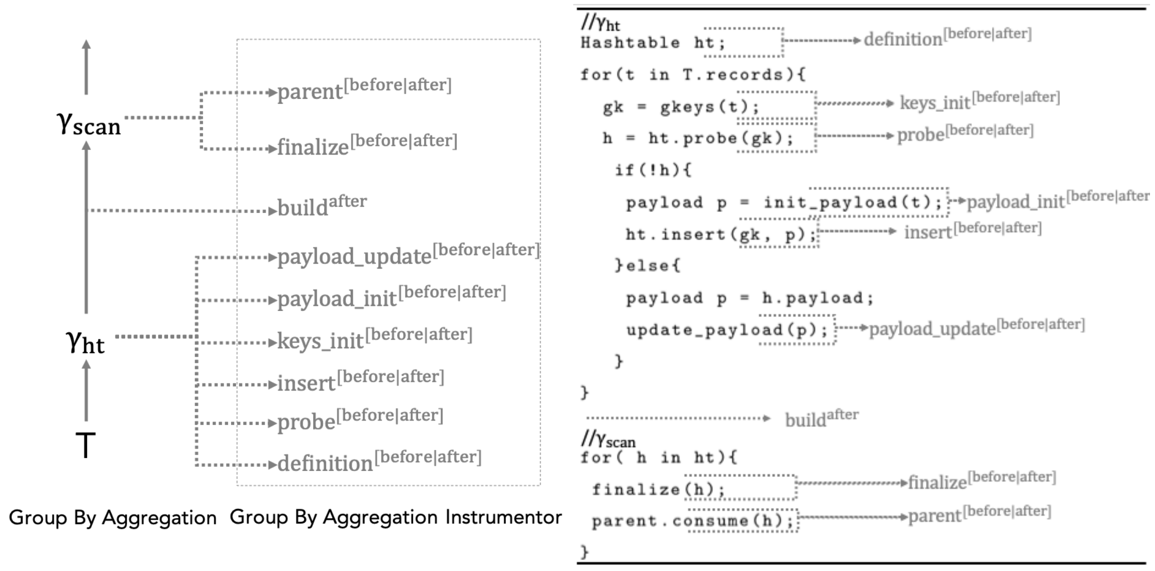


```
for(t in T.records){
 if( pred(t) ){
   // σ_P^before
   parent.consume(t);
   // σ_P^after
 }else{
   // σ_N
 }
}
```

Figure 6.4: Physical plan (left) and corresponding source code (right) for the selection operator under instrumentation.

$\sigma_P^{\mathtt{after}}$ corresponds to the code fragment that will be executed for records that satisfy the selection but only after the parent operator has consumed each record. Finally, $\sigma_N$ corresponds to the code fragment that would be executed for records that did not satisfy the selection. (Note that $\sigma_N$ does not exist under normal query execution. Rather SMOKE will introduce it if applications need to integrate their logic in this fragment.)

Now, SMOKE further associates each instrumentation point with data flows that applications can use in their instrumentation logic. For the case of selection, $\sigma_P^{\mathtt{before}}$ and $\sigma_P^{\mathtt{after}}$ are associated with the stream of records that satisfy the predicate, while $\sigma_N$ is associated with the stream of records that do not satisfy the predicate. To enable this functionality, as we will see in Section 6.4 in more detail, such data flows can be passed as parameters to functions corresponding to instrumentation points so that applications can devise their instrumentation logic based on them. For instance, a negative provenance manager can capture the records that did not satisfy the selection by operating on the underlying data flow of $\sigma_N$. It is important to note, however, that applications may not consider the underlying data flows in their logic. For instance, a monitoring application, similar to the one that we introduced in Section 6.2.2, can take time statistics on the parent consumption of the selection operator by pushing its logic in $\sigma_P^{\mathtt{before}}$ and $\sigma_P^{\mathtt{after}}$ without processing the stream of records that satisfy the selection.



```
//γht
Hashtable ht;
for(t in T.records){
  gk = gkeys(t);
  h = ht.probe(gk);
   if(!h){
    payload p = init_payload(t);
    ht.insert(gk, p);
   }else{
    payload p = h.payload;
    update_payload(p);
   }
}
//γscan
for( h in ht){
 finalize(h);
 parent.consume(h);
}
```

Figure 6.5: Physical plan (left) and corresponding source code (right) for the hash-based groupby aggregation operator under instrumentation.

## 6.3.2 Hash-based Group-By Aggregation

Similarly to selections, we can break the logic of hash-based group-by aggregations in code fragments and associate them with instrumentation points. The hash-based group-by aggregation is split into two operators: $\gamma_{\mathtt{ht}}$ and $\gamma_{\mathtt{scan}}$. $\gamma_{\mathtt{ht}}$ constructs the hash table that keeps for every group in the input a payload that keeps track of (the partial state of) aggregations. $\gamma_{\mathtt{scan}}$ scans the hash table, finalizes the aggregations, and emits results to its parent for consumption. Hence, each operator can be further decomposed in code fragments that implement the logic of the group-by aggregation. In turn, SMOKE provides instrumentation points on the underlying code fragments. For completeness, the instrumentation points for the hash-based group-by aggregations are described in Table 6.1; Figure 6.5 shows these points on the physical plan for group-by aggregation and as annotations in the corresponding source code. The design principles behind them are similar to the ones of selections (i.e., each key operation in the logic of group-by aggregation is wrapped with before and after instrumentation points and associated with data flows).

One important difference from selections, however, is that selections are stateless operators while hash-based group-by aggregations maintain state (i.e., hash tables) that instrumentors need to access. For instance, INJECT provenance capture on group-by aggregations needs to access state to append rid arrays in the intermediate state maintained for each group in the hash table, as we showed in Section 3.4.3. To enable this functionality, SMOKE allows applications to operate on hash tables, similar to how SMOKE allows applications instrumenting selections to operate on underlying data flows. More specifically, SMOKE exposes instrumentation points in the logic of group-by aggregation and associates them with information related to the hash table (as opposed to data flow in the case of selection).

Consider the source code of the group-by aggregation in Figure 6.5(right). Operations on hash tables include their definition, insertion of a new (key, payload), update of the payload, probing, and finalizing. In turn, as illustrated in Figure 6.5(right), SMOKE introduces instrumentation points before and after the definition of hash tables, insertion of new (key, payload) entries, initialization of keys, initialization of payloads, update, and finalizing of payloads. Using these points, applications can alter the underlying hash table by adding, deleting, or altering keys, payloads, and the overall structure of the hash tables. To do so, SMOKE associates each point with hash table related information, similarly to how we associated data flows with instrumentation points of selection operators.

More precisely, $\texttt{definition}^{\texttt{[before|after]}}$ are associated with the hash table definition to allow applications to add; remove; or modify keys and payload definition. (We discuss these operations in more detail in Section 6.8). $\texttt{keys\_init}^{\texttt{before}}$ and $\texttt{keys\_init}^{\texttt{after}}$ are associated with the record used to construct the key and the initialized keys, respectively. $\texttt{probe}^{\texttt{after}}$ is associated with whether the probe failed or not while $\texttt{probe}^{\texttt{before}}$ is associated with the keys to probe the hash table. $\texttt{payload\_init}^{\texttt{before}}$ and $\texttt{payload\_init}^{\texttt{after}}$ are associated with the record used to initialize the payload and the initialized payload. $\texttt{insert}^{\texttt{[before|after]}}$ are associated with keys and payload that are inserted in the hash table. $\texttt{update}^{\texttt{[before|after]}}$ are associated with the payload to update and the updated payload, respectively. Finally, `finalize`$^{\texttt{[before|after]}}$ are associated with the hash table entry before and after the finalize step.

To illustrate these points, lets us consider the complicated scenario of provenance capture with the INJECT approach that we presented in Section 3.4.3.

**Example 8 (INJECT Provenance Capture on Group-By Aggregation)** The INJECT approach for provenance capture adds an rid array to the payload of each group where it stores the rids of the record that contributed to the group. After building, during the scan phase. To implement this functionality using the instrumentation points above we can do the following: First, we need to alter the definition of the hash table to add the rid array as another attribute of the payload. We can do so either before or after the hash table has been defined (`definition`$^{\texttt{[before|after]}}$). Then, whenever we insert a new (key, payload) to the hash table we need to create a new rid array and add it in the payload. We can do so right after the initialization of the payload by the group-by aggregation (`payload_init`$^{\texttt{after}}$). On update of the payload, we also need to append the rid of the current input record to the rid array that we added to the payload during insertion. We can do so, before or after the group-by aggregation updates the payload (`update`$^{\texttt{[before|after]}}$). Finally, during the scan phase of the group-by aggregation, we need to get each rid arrays to add it to the backward rid list. We can do so before or after the finalize step (`finalize`$^{\texttt{[before|after]}}$).

### 6.3.3 Joins

Instrumentation points in the logic of hash-based and nested loop joins follow the design principles that we discussed so far. The complete list of instrumentation points along with their description for both join operators is in Table 6.1. Here, we focus on the problem of associating instrumentation points in the logic of joins with records that did not contribute to any join result. Essentially, similarly to how $\sigma_N$ provides access on the data flow of records that did not satisfy the selection predicate, here we seek to gain access on data flows for records that did not satisfy the join predicate.

A straightforward way to accomplish this functionality is by using the instrumentation points of the join operators. In fact, in Section 6.2.2, where we described the negative provenance manager we did precisely that. However, instead of shifting the burden to applications for gaining access to such data flows, SMOKE provides direct access to them by implementing the underlying logic on its own. This highlights the powerful concept of compositionality behind instrumentation: common-place instrumentation logic and instrumentation practices can be encapsulated into the database and exposed as new instrumentation primitives.

Next, we discuss how SMOKE implements the underlying logic and exposes instrumentation points so that applications can gain access on negative data flows of joins.

### Hash-based Joins

We first discuss how to gain access to the records from the left side that did not contribute to any join result and then to the ones from the right side.

**Left side.** Records from the build (left) side that did not contribute to any join results are only known after the join has been executed. This is because we can only be certain whether a record from the build (left) side of the join has contributed to a join result only after we have probed the underlying hash table with every record from the right (probe) side. To provide access to records from the probe (right) side that did not satisfy the join predicate, SMOKE instruments the hash table of the join to add a bit in the payload of each key. (Recall from our discussion on instrumentation points on group-by aggregation how we can alter hash tables in such a way.) When the build has finished, the bit is set to 0 for all keys. Whenever a probe from the right side succeeds the bit is turned to 1. After the join execution, SMOKE scans the hash table and emits the records that did not satisfy the join on the instrumentation point `not_joined_from_build`. It does so by checking which keys have the bit set to 1. Finally, note that if the hash table join does not contain the full record, SMOKE will also append the corresponding rids during building the hash table, so that they can be accessed from `not_joined_from_build`.

**Right side.** Records from the probe side (right) that did not contribute to any join result are easier to access. This is because after we probe a hash table we know whether the record from the right side matches or not. Hence, SMOKE introduces a `not_joined_from_probe` point that corresponds to the code fragment where the probe fails.

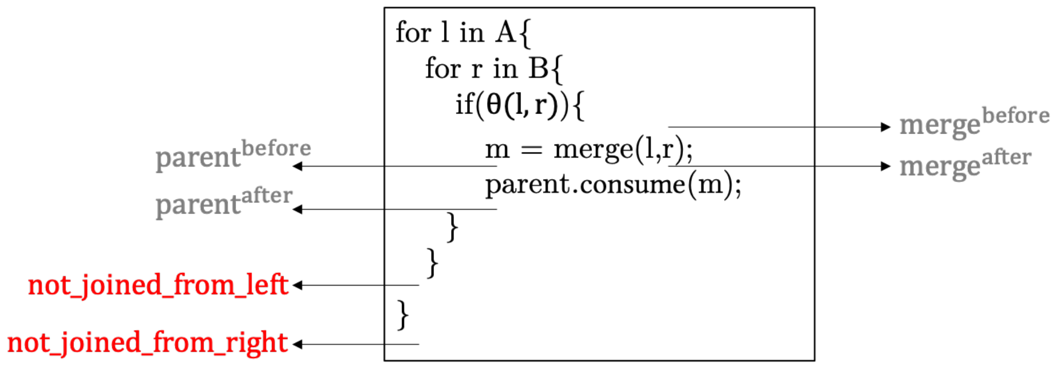


Figure 6.6: Source code for the nested loop join under instrumentation.

## Nested Loop Joins

**Left side.** For a record from the left side of a nested loop join, we know if it does not contribute to any join result when we finish searching for matches for that record on the right side. SMOKE keeps track whether a match was found and if not it exposes the left record on the `not_joined_from_left` instrumentation point (see Figure 6.6) that corresponds to the point right after finishing a loop on the right side. (Note that this case is exactly the example on negative provenance capture that we discussed in Section 6.2.2.) SMOKE keeps tracks whether a match was found for each left record by updating a flag after the parent of the join has consumed the join result (see `parent`$^{\text{after}}$ in Figure 6.6).

**Right side.** With regards to records from the right side, we can only be sure about which ones have not contributed to a join result after we have finished the nested loop join execution. SMOKE provides access to these records similarly to how it provides access to records from the left side of a hash-based join that did not contribute to a join result. More specifically, we keep a bit per record of the right side indicating whether or not it has contributed to a join result. After the execution of the nested loops, we check the bit per record of the right side and emit the ones that have not contributed to the join on the instrumentation point `not_joined_from_right` (see Figure 6.6).

## 6.3.4 Other Operators

So far we have introduced the main design principles behind the introduction of instrumentation points on selections, group-by aggregations, and joins that SMOKE supports. Other operators in SMOKE's physical algebra have been introduced following the same principles. To avoid redundancy and to ease our presentation we omit further discussion on other operators. For completeness, the instrumentation points for every operator as currently supported in SMOKE are described in Table 6.1. Finally, an important note is that the same principles are also applicable for pipelines and plans (i.e., SMOKE introduces instrumentation points before and after pipelines and plans and associates them with data flows such as intermediate, input, and output relations).

Next, we introduce how SMOKE allows applications to implement Instrumentors to push their instrumentation logic in the instrumentation points that we discussed in this section.

Table 6.1: Instrumentation points provided by SMOKE in the logic of individual physical operators, pipelines, and plans.

| Abbr. | Instrumentation Point | Short description |
|---|---|---|
| | SELECTION | |
| $\sigma_P^{\texttt{before}}$ | `before_parent` | Before the parent of the selection consumes a record that satisfied the selection. |
| $\sigma_P^{\texttt{after}}$ | `after_parent` | After the parent of the selection has consumed a record that satisfied the selection |
| $\sigma_N$ | `not_satisfied` | Whenever a record does not satisfy the selection. |

HASH-BASED GROUP-BY AGGREGATION

| | | |
|---|---|---|
| $\gamma\mathtt{ht}^{\mathtt{before}}_{\mathrm{probe}}$ | `before_probe` | Before probing a hash table on grouping keys. |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathrm{probe}}$ | `after_probe` | After probing a hash table on grouping keys. |
| $\gamma\mathtt{ht}^{\mathtt{before}}_{\mathrm{insert}}$ | `before_insert` | Before inserting (group keys, payload) which happens in case probing fails |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathrm{insert}}$ | `after_insert` | After inserting (group keys, payload) which happens in case probing fails |
| $\gamma\mathtt{ht}^{\mathtt{before}}_{\mathtt{def}}$ | `after_definition` | After the definition of the hash table. |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathtt{def}}$ | `before_definition` | Before the definition of the hash table. |
| $\gamma\mathtt{ht}^{\mathtt{before}}_{\mathtt{keys\_init}}$ | `before_keys_init` | Before the initialization of the grouping keys. |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathtt{keys\_init}}$ | `after_keys_init` | After the initialization of the grouping keys. |
| $\gamma\mathtt{ht}^{\mathtt{before}}_{\mathtt{payload\_init}}$ | `before_payload_init` | Before the initialization of the payload for a new hash table entry. |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathtt{payload\_init}}$ | `after_payload_init` | After the initialization of the payload for a new hash table entry. |
| $\gamma\mathtt{ht}^{\mathtt{before}}_{\mathtt{payload\_update}}$ | `before_payload_update` | Before updating the payload for a hash table entry. |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathtt{payload\_update}}$ | `after_payload_update` | After updating the payload for a hash table entry. |
| $\gamma\mathtt{ht}^{\mathtt{after}}_{\mathtt{build}}$ | `after_ht_build` | After finishing building the hash table but before scanning for finalizing aggregations. |
| $\gamma\mathtt{scan}^{\mathtt{before}}_{\mathtt{finalize}}$ | `before_finalize` | Before finalizing the aggregations on a hash table entry. |

| | | |
|---|---|---|
| $\Upsilon\mathtt{scan}^{\mathtt{after}}_{\mathtt{parent}}$ | `after_finalize` | After finalizing the aggregations on a hash table entry but before sending to the parent. |
| $\Upsilon\mathtt{scan}^{\mathtt{after}}_{\mathtt{parent}}$ | `after_ht_parent` | After sending to parent. |

HASH-BASED JOIN

| | | |
|---|---|---|
| * | * | Hash-based joins share similar instrumentation points with the hash-based group-by aggregation for defining, probing, and updating a hash table. |
| $\bowtie^{\mathtt{after}}_{\mathtt{merge}}$ | `after_merge` | Right after merging two joined tuples from the two sides but before the parent consumes the merged result. |
| $\bowtie^{\mathtt{before}}_{\mathtt{parent}}$ | `before_join_parent` | Before the parent has consumed a join result. |
| $\bowtie^{\mathtt{after}}_{\mathtt{parent}}$ | `after_join_parent` | After the parent has consumed a join result. |
| $\bowtie^{\mathtt{probe}}_{\mathrm{N}}$ | `not_joined_from_probe` | Whenever a probe fails to find a match. Used to find the records from the probe (right) side that did not satisfy a join predicate. |
| $\bowtie^{\mathtt{build}}_{\mathrm{N}}$ | `not_joined_from_build` | Point introduced by SMOKE for consumption of all records from the build (left) side that did not satisfy a join predicate. |

NESTED-LOOP JOIN

| | | |
|---|---|---|
| $\bowtie^{\mathtt{before}}_{\vartheta}$ | `before_join_predicate` | Right before the join predicate. |
| $\bowtie^{\mathtt{before}}_{\mathtt{merge}}$ | `after_merge` | Right before merging two joined tuples from the two sides but before the parent consumes the merged result. |

| | | |
|---|---|---|
| $\bowtie^{\texttt{after}}_{\texttt{merge}}$ | `after_merge` | Right after merging two joined tuples from the two sides but before the parent consumes the merged result. |
| $\bowtie^{\texttt{before}}_{\texttt{parent}}$ | `before_parent` | Before the parent has consumed a join result. |
| $\bowtie^{\texttt{after}}_{\texttt{parent}}$ | `after_parent` | After the parent has consumed a join result. |
| $\bowtie^{\texttt{right}}_{\mathrm{N}}$ | `not_joined_from_right` | Point introduced by SMOKE for consumption of all records from the right side of the join that did not satisfy a join predicate. In contrast, to the hash-based join instrumentor, records from the right side (or probe side for hash-based joins) are known only after the nested loop join has finished. |
| $\bowtie^{\texttt{left}}_{\mathrm{N}}$ | `not_joined_from_left` | Point introduced by SMOKE for consumption of all records from the build side that did not satisfy a join predicate. In contrast to the hash-based joined instrumentors, records from the build side that did not contribute to a join result are available when we finish the loop on the right (probe) side |
| $\bowtie^{\texttt{right}^{\texttt{before}}}_{\mathrm{N}}$ | `before_right` | Point right before the inner loop |
| $\bowtie^{\texttt{right}^{\texttt{after}}}_{\mathrm{N}}$ | `after_right` | Point right after the inner loop. Same as `not_joined_from_right` but not associated with the negative data flow. |

| | | |
|---|---|---|
| **CROSS PRODUCT** | | |
| $\times^{\texttt{before}}_{\texttt{merge}}$ | `after_merge` | Right before merging two joined tuples from the two sides but before the parent consumes the merged result. |
| $\times^{\texttt{after}}_{\texttt{merge}}$ | `after_merge` | Right after merging two joined tuples from the two sides but before the parent consumes the merged result. |
| $\times^{\texttt{before}}_{\texttt{parent}}$ | `before_parent` | Before the parent has consumed a join result. |
| $\times^{\texttt{after}}_{\texttt{parent}}$ | `after_parent` | After the parent has consumed a join result. |
| $\mathrm{P}^{\texttt{after}}_{\texttt{end}}$ | `after_plan_end` | Right after a plan ends executing. |
| **MATERIALIZATION** | | |
| $\mathrm{M}^{\texttt{before}}_{\texttt{materialization}}$ | `before_materialization` | Before the materialization of a record. |
| $\mathrm{M}^{\texttt{after}}_{\texttt{materialization}}$ | `after_materialization` | After the materialization of a record. |
| **PIPELINE** | | |
| $\vdash^{\texttt{before}}_{\texttt{start}}$ | `before_pipeline_start` | Right before a pipeline starts. |
| $\vdash^{\texttt{after}}_{\texttt{end}}$ | `after_pipeline_end` | Right after a pipeline ends. |
| **PLAN** | | |
| $\mathrm{P}^{\texttt{before}}_{\texttt{start}}$ | `before_plan_start` | Right before a plan starts executing. |
| $\mathrm{P}^{\texttt{after}}_{\texttt{end}}$ | `after_plan_end` | Right after a plan ends executing. |

## 6.4 Instrumentation Logic

In the previous section, we introduced points in the logic of physical operators that overall comprise the instrumentation semantics that SMOKE exposes to applications for integrating their logic within operators. Building on this semantics, the actual implementation of the instrumentation logic happens within instrumentors (i.e., interfaces that expose programmatic access to instrumentation points)—which are the focus of this section.

More specifically, next we start by explaining what are instrumentors in SMOKE, their connections with instrumentation points, and their main types (Section 6.4.1). Then, we introduce in detail the different types of instrumentors that allow the specification of the instrumentation logic either imperatively (Section 6.4.2) or, in some cases, declaratively (Section 6.4.3). Finally, to close our discussion on instrumentors and the instrumentation process, we present capabilities that SMOKE provides for instrumenting instrumentors (Section 6.4.4) as well as how applications register instrumentors to physical operators so that physical operators can execute the instrumentation logic during execution (Section 6.4.5).

```
class SelectionInstrumentor{
  virtual void on_before_parent(...);
  virtual void on_after_parent(...);
  virtual void on_not_satisfied(...);
};
```

Figure 6.7: Instrumentor of selection for imperative specification of the instrumentation logic.

## 6.4.1 Instrumentors and Instrumentation Points

An instrumentor of a physical operator is an interface that exposes functions that applications implement to push their logic within a physical plan. An important distinction between instrumentors in SMOKE is on how applications implement their logic within instrumentors. SMOKE provides two such modes and corresponding instrumentation: imperative and in some cases declarative. Sections 6.4.2 and 6.4.3 provide more details on them. Here, we discuss their connections with instrumentation points.

**Imperative.** For imperative specification, functions exposed by instrumentors correspond to the instrumentation points of the physical operator, and we refer to them as *instrumentation functions*. For instance, consider the `SelectionInstrumentor` in Figure 6.7 which is an instrumentor that allows imperative specification of the instrumentation logic on the selection operator. The instrumentation functions `on_before_parent`, `on_after_parent`, and `on_not_satisfied` correspond to the points $\sigma_P^{\texttt{before}}$, $\sigma_P^{\texttt{after}}$, and $\sigma_N$.

**Declarative.** For declarative specification, recall that some instrumentation points are associated with data flows. Each data flow can be treated as a streams of tuples and, as such, points can be treated as sources of tuples from where instrumentors can consume from. This allows applications to express their instrumentation logic by using SQL, if this is possible, and instrumentors only expose functions for registering SQL queries. Finally, we note that SMOKE under the cover compiles the SQL specification into imperative instrumentors.

Next, we describe in more detail the two types of instrumentors (and their subtypes).

## 6.4.2 Imperative Specification

We start by describing instrumentors that allow applications to specify their logic imperatively. Imperative instrumentors are further subdivided into interpretation- and compilation-based ones, following similar semantics with interpretation and compilation-based implementation of physical operators in databases. Recall that databases perform either interpretation- or compilation-style plan execution each with unique performance characteristics, targeting different workload types [KLK+18]. Similarly, SMOKE provides interpretation- and compilation-based instrumentors that result in interpretation- and compilation-style execution of the instrumentation logic to account for different instrumentation workload types.

The main difference between interpretation and compilation-based instrumentors lies in how applications implement their instrumentation functions. Next, we discuss both types of instrumentors. As an example to drive our discussion, we will consider the implementation of provenance capture on the selection operator.

### 6.4.2.1 Interpretation-based Instrumentors

Consider the interpretation-based instrumentor for provenance capture on the selection operator in Figure 6.8. Interpretation-based instrumentors are C++ objects that implement the interface of interpretation-based instrumentors of individual operators, pipelines, and plans.

```
class ProvenanceCapture :
public InterpretedSelectionInstrumentor{
 RID[] bw;
 RID coid=0, csize=10;
 Smoke& db;
 ProvenanceCapture(Smoke& d, Selection& op)
 :db(d),InterpretedSelectionInstrumentor(op){
   bw = db.storage.newRIDArray(csize, "bw_sel");
 }
 void on_after_parent(Record t){
   if(coid+1 == csize) realloc(...);
   bw[coid++] = t.rid;
 }
};
```

```
ProvenanceCapture instr;
for(t in T.records){
 if( pred(t) ){
  parent.consume(t);
  instr.on_after_parent(t); // σP^after
 }
}
```

Figure 6.8: Provenance capture on selections with interpretation-style specification of the instrumentation logic (left) and corresponding source code generated after compilation (right).

For instance, `ProvenanceCapture` in our example extends the interpretation-based instrumentation interface of the selection operator `InterpretedSelectionInstrumentor` by implementing the on_after_parent(Record) function. Given an instrumented physical plan with physical operators instrumented as in our example, SMOKE will compile it into source code such as the one shown in Figure 6.8(right).

Interpretation-based instrumentors will first be initialized and the implemented instrumentation functions will be called (e.g., as shown in the source code in Figure 6.8) during execution to pass the control flow from the plan execution to instrumentors. SMOKE initializes instrumentors by calling their corresponding constructor with a reference to `Smoke` and the operator that they instrument. It does so to enable instrumentors to use the underlying components of SMOKE as well as guide their logic based on the operator they instrument. In our example in Figure 6.8, `ProvenanceCapture` during initialization uses SMOKE's storage to create a new array of rids and set its name as "bw_sel". Then, at runtime, the function on_after_parent(Record) of the initialized `ProvenanceCapture` object `instr` will be called every time the parent returns from its consumption. (Note that the call `parent.consume` in Figure 6.8 is syntactic sugar. In practice, SMOKE will inline the whole consumption logic of the parent within the source code. In contrast, the code for `instr.on_after_parent(t);` will not be inlined within the source code by SMOKE.)

The problem with interpretation-based instrumentors lies on the fact that each call to the instrumentation function is a virtual function call which impacts performance. (Recall that we showed experimentally the impact of virtual function calls on provenance capture in Section 3.7.1.) To avoid this problem, an alternative that allows the instrumentation logic to be tightly integrated with the query execution is through compilation-based instrumentors.

```
class ProvenanceCapture :
public CompiledSelectionInstrumentor{
 RidArrayVariable bw;
 RidVariable coid, csize;
 Smoke& db;

 ProvenanceCapture(Smoke& d, Selection& op)
 :db(d),CompiledSelectionInstrumentor(op){
   Compiler& c = smoke.compiler;
   coid = c.newRIDVariable();
   csize = c.newRIDVariable(10);
   bw = c.newRidArrayInStorage(csize,
                            "bw_sel");
 }
 void on_after_parent(RecordVariable t){
   Compiler& c = smoke.compiler;
   Condition cond; CodeBlock b;
   cond = c.newCondition(coid+1==csize);
   b = c.newCodeBlock("realloc(...)");
   c.newIFBLock(cond, b);
   c.makeAssignment(bw[coid++], t.rid);
 }
};
```

```
RID coid=0,csize=10;
RID[] bw = db.storage.newRIDArray(csize,
                                  "bw_sel");
for(t in T.records){
 if( pred(t) ){
   parent.consume(t);
   if(coid+1 == csize) realloc(...);
   bw[coid++] = t.rid;
 }
}
```

Figure 6.9: Provenance capture on selections with compilation-based specification of the instrumentation logic (left) and corresponding source code generated after compilation (right).

#### 6.4.2.2 Compilation-based Instrumentors

Consider the compilation-based instrumentor for provenance capture on the selection operator in Figure 6.9. Similarly to interpetation-based instrumentors, compilation-based instrumentors are also C++ objects. This time, however, they implement the compilation-based instrumentor interface. While both types of instrumentors share the same interface, the compilation-based interface differs in terms of what each instrumentation point takes as input and how applications can express their instrumentation logic.

Compilation-based instrumentors express their instrumentation logic in the internal IR of SMOKE that we discussed in Chapter 2. Recall, that the IR corresponds to an AST with nodes defining function, conditional, loop, and code blocks. For expressing logic within blocks, SMOKE provides a type system involving compilation-based interfaces to primitives (e.g., int, double precision, string) and data structures such as hash tables, vectors, arrays, scalars, records, and tables. Logic expressed in this intermediate representation is compiled to C++ source code by SMOKE's compiler and to binary using g++. Hence, instrumentors can also inline their C++-specific code within blocks. We anticipate future work on SMOKE to compile its internal IR into targets either than C++ (e.g., to LLVM or python) so that instrumentors can use in their logic the runtimes and capabilities of other target IRs.

To illustrate how SMOKE compiles the instrumentation logic within query plans, consider the provenance capture selection instrumentor and corresponding source code post-compilation by SMOKE in Figure 6.9. During initialization, the provenance capture instrumentor defines its state. To do so, it constructs the RID variables `csize` and `coid` as well as the RID array `bw` by calling newRIDVariable and newRidArrayInStorage(csize, "bw_sel"), respectively. SMOKE compilation of the resulting IR for the initialization results in the first two lines of the source in Figure 6.9 that will initialize at runtime the RID and RID array variables. Similarly, `ProvenanceCapture` defined the logic of on_after_parent in SMOKE's IR. The result of the compilation of on_after_parent essentially inlines the logic of provenance capture, that we showed with the interpretation-based instrumentor, into the physical plan to avoid the virtual function calls of the interpretation-based instrumentors. Finally, note the difference between the two types of instrumentors in terms of input parameters of on_after_parent. The interpretation-based was getting as input a `Record` whereas the compilation one takes as input a `RecordVariable`. Similarly to database management systems, interpretation-based instrumentors will have to pay the cost of interpreting records at runtime whereas with the compilation-based way the logic over records is defined at compile time and inlined to avoid further interpretation costs.

#### 6.4.2.3 Advanced Compilation-based Instrumentors

The instrumentors we have presented so far, while adequate for simple instrumentation tasks, they are limited in two ways. First, they are strongly tied to the parameters that instrumentation functions take as input. Second, for complicated logic where we want to connect instrumentors or introduce our own instrumentation operators (e.g., similarly to how we did with provenance capture in Chapter 3), the types of instrumentors presented so far provide little flexibility (i.e., instrumentation applications need to implement their own operators and connect them in their own ways).

To address these issues, SMOKE also introduces an advanced compilation-based instrumentor that, while it requires a more sophisticated development of the instrumentation logic, it is more flexible than the previous types of instrumentors. The main idea behind this type of instrumentor is that each instrumentation point can be considered as a source from where instrumentors can consume from. This idea forms the basis for introducing physical instrumentation operators similarly to how databases implement physical operators under the producer-consumer model, as we discussed in Chapter 2.

Consider our example on provenance capture on selection again. Instead of having the on_after_parent as a function, the advanced compilation-based instrumentor considers it as an operator. Similar, to other physical operators under the producer-consumer model, on_after_parent has a `consume` and a `produce` function associated with it. By calling the `produce` function on on_after_parent we can request from the physical plan to produce a piece of information at the instrumentation point $\sigma_P^{\texttt{after}}$. Note that this information may not be a record but rather whatever is requested that either physical operators or other instrumentation operators of the physical plan can produce. In the producer-consumer compilation model this can be accomplished by asking an operator (e.g., on_after_parent) to produce whatever is specified in a `Required` variable. In our example, the `Required` variable could specify that we only want the rid of the input record without caring about the record itself. (This decouples us from the fixed parameters of the other types of instrumentors). Then, when the compilation reaches the instrumentation point it can produce the `Required` information and let the parent of on_after_parent to consume it.

A more intuitive way to understand the notion of `Required` in contrast to the fixed parameters of other operators is through the semantics of a point in a source code. So far, we have tied each point to particular information that it can be processed at that point. However, from a source code perspective, the information available at a point is whatever is in the stack during the execution at this point. As a result, `Required` can be anything that could appear in the stack. For more background on how `Required` works under the producer-consumer compilation model refer to our background in Chapter 2.

Now, by exposing a unifying producer-consumer framework both for instrumentors and physical operators we can construct sophisticated plans where each node has the same compilation and execution interface. This has a nice side-effect that it simplifies the compilation process. Note that for compilation purposes, the other types of operators are also treated as being compliant with producer-consumer interfaces. This is because the fixed input parameters are essentially an instance of what `Required` can be, instrumentation functions are essentially consumers of this instance of `Required` information, and constructors of either interpretation or compilation-based interfaces serve as `produce` functions.

## 6.4.3 Declarative Specification

Whether instrumentation applications implement their logic in an interpreted or compilation style, they still need to implement their logic imperatively which is time-consuming, error-prone, and may lack optimizations. In this direction, SMOKE allows instrumentation applications to express their logic declaratively in SQL terms.

The main idea is that instrumentation functions, in most cases, take us input records from instrumentation points of a query plan. To this end, if the instrumentation logic involves processing of records then, in many cases, this logic may be expressible in SQL terms. To enable this functionality, SMOKE allows instrumentors to consume instrumentation points with SQL by registering SQL queries with the function `consume_with_SQL` which is part of the interface of instrumentors.

```
class ProvenanceCapture :
public CompiledSelectionInstrumentor{
 ...
 void consume_with_SQL(){
   register("V1 = SELECT * "
   " FROM on_not_satisfied");
 }
};
```

```
RID coid=0,csize=10;
RID[] bw = db.storage.newRIDArray(csize,
                                  "bw_sel");
View V1;

for(t in T.records){
 if( pred(t) ){
   parent.consume(t);
   if(coid+1 == csize) realloc(...);
   bw[coid++] = t.rid;
 }else{
   V1.append(t);
 }
```

Figure 6.10: Declarative specification of negative provenance capture on selections (left) and source code generated by SMOKE after compilation (right).

For instance, consider the `ProvenanceCapture` instrumentor of a selection operator in Figure 6.10. This `ProvenanceCapture` instrumentor extends the compilation-based `ProvenanceCapture` instrumentor that we presented in Figure 6.9 by implementing the function `consume_with_SQL`. The function `consume_with_SQL` registers the query `V1 = SELECT * FROM on_not_satisfied` which will materialize all the tuples that did not satisfy the selection in view `V1`. When SMOKE's compiler is presented with such registrations, it will first convert them into equivalent compilation-based instrumentation functions. In turn, the resulting instrumentation functions will be compiled to source code similarly to how instrumentation- and compilation-based instrumentors are compiled to source code. (Section 6.9 discusses in more detail the overall compilation process.) The resulting source code for our example is shown in Figure 6.10(right) with red lines corresponding to the instrumentation logic for the materialization of `V1`. blue lines correspond to positive provenance capture, and we include them in the source code to highlight that applications can implement both imperative and declarative instrumentors at the same time.

Finally, it should be noted that not every instrumentation logic is expressible in SQL terms. For one, not every function exposed by the instrumentation framework of SMOKE takes as input records (e.g., instrumentation points on hash tables). Furthermore, even functions that take as input records may not be able to express their logic in SQL. For instance, as we will see with physical database designers in Chapter 10, instrumentors take as input records in order to induce new or restructure the current physical database design. Yet, we believe that there is ample space for future work so that declarative specifications of the instrumentation logic can be compiled down into optimized instrumentors. In this direction, in Part II we present several instrumentation techniques to expose best instrumentation practices across domains that future declarative specifications could be compiled into.

### 6.4.4 Instrumenting Instrumentors

Our discussion above on declarative specification of instrumentors introduces us to a central principle of SMOKE. That of recursive instrumentation or instrumentation of instrumentors. Consider again view `V1` in our example above. This is a SQL query and, as such, we can instrument it similarly to how we can instrument every query that enters the SMOKE. In other terms, and besides only instrumentors expressed in SQL terms, an instrumentor can be instrumented as well by other instrumentors. This is a powerful construction because it enables several instrumentation capabilities. For instance, as we will see in Section 6.5, SMOKE internally instruments instrumentors to get runtime statistics (e.g., memory pressure and CPU consumption) in order to notify the same or other instrumentors about events (e.g., the memory pressure or CPU consumption exceeded a limit).

To enable this functionality, SMOKE's framework follows the logic of instrumentation of physical plans. This time, instead of instrumenting a physical plan, however, instrumentors instrument instrumented physical plans. The main idea behind this functionality is that instrumentation functions can be treated as operators that can expose their own instrumentation points or, in cases when the instrumentation logic uses the underlying constructs of SMOKE (e.g., hash tables), they can use the instrumentation points provided by SMOKE.

Currently, SMOKE supports only the latter. How instrumentors can introduce their own instrumentation points is beyond the scope of this work.

### 6.4.5 Registration Process

So far we have covered how an application can implement the instrumentor of a physical operator. This is the first step of the instrumentation process. What is left is that the application needs to register the instrumentor to the physical operator so that the physical operator can execute the instrumentation logic during execution.

We illustrate the registration process of imperative selection instrumentors. (For other operators the process is similar). Recall that, given a selection operator, applications can implement their logic for $\sigma_{\mathrm{P}}^{\mathtt{before}}$, $\sigma_{\mathrm{P}}^{\mathtt{after}}$, and $\sigma_{\mathrm{N}}$ by implementing the functions `on_before_parent`, `on_after_parent`, and `on_not_satisfied`, respectively.

To register the selection instrumentor, the selection operator exposes three function calls (i.e., `register_before_parent`, `register_after_parent`, and `register_not_satisfied`). Each of these function calls take as input a selection instrumentor which is appended on a corresponding array of instrumentors in the selection operator (i.e., `before_parent[]`, `after_parent[]`, and `not_satisfied[]`) to account for cases when multiple applications register their logic. Finally, during execution of the instrumented physical plan, the selection operator will execute the corresponding functions of the instrumentors based on what instrumentation points they have registered on.

Note that if the instrumentation logic has been specified declaratively, SMOKE will compile it into imperative instrumentors. Hence, the registration process is the same but handled by SMOKE. Also, if the instrumentor is a compilation-based one, note that the physical operator will not execute in practice the instrumentation functions because these functions will be inlined in the source code of the physical operator due to instrumentation-aware compilation process of SMOKE.

So far we have covered the first component of the Physical Plan Instrumentation Framework of SMOKE (i.e., Instrumentors and their instrumentation Points). Next, we continue with the introduction of the remaining components the framework, starting from Scheduler.

## 6.5 Scheduler

The Scheduler component of the instrumentation framework is responsible for scheduling the instrumentation logic relative to operators as well as instrumentors relative to one another.

**Challenges.** There are two main challenges regarding the scheduling of instrumentation logic. First, instrumentation points allow instrumentation logic to be injected into query plans. This results in overheads that some application domains (e.g., interactive applications) may not tolerate. If that's the case, then instrumentors should be able to either defer their whole logic after execution or partially inject and partially defer parts of their logic if they can tolerate some overhead on query execution. Second, note that multiple instrumentors can instrument a single instrumentation point. In such cases, SMOKE needs to identify an execution order of the different instrumentors.

**Solutions Overview.** To address the first challenge, the Scheduler provides automatic ways to defer the instrumentation logic after the query execution in many cases. The main idea behind automatic defer is that (positive or negative) data flows that instrumentation points correspond to can be regenerated after query execution for instrumentation purposes. Besides automatic ways, the Scheduler also provides functions that applications can use for scheduling the instrumentation logic on their own. To address the second challenge, SMOKE automatically detects if there is any internal dependency between instrumentors and automatically picks an order for instrumentors per instrumentation point. If dependencies are found, however, applications are responsible for providing an ordering of instrumentors. In this direction, the Scheduler exposes ordering functions that applications can use to provide an order of the instrumentors.

### 6.5.1 Automatic Defer

The main idea behind automatically deferring instrumentation logic is that the same instrumentors that one can define for injection in many cases can also be used under deferred application of the instrumentation logic. This is possible by first executing an operator without instrumentation. After finishing its execution we can re-execute the operator, or parts of its, this time applying the instrumentor functions. Next, we first discuss on an example to better illustrate this logic. Then, we discuss how SMOKE re-executes operators, in an efficient manner, if possible, to apply the instrumentation logic. (Finally, note that SMOKE currently provides automatic defer functionality only for compilation-based instrumentors because SMOKE needs to detect dependencies and guarantee semantics of operators, which are simple operations if the logic is expressed in the IR of SMOKE. Automatic defer for interpretation-based instrumentors is left for future work.)

```
for(t in T.records)
 if( pred(t) )
   parent.consume(t);

coid=0;
for(t in T.records){
 if( pred(t) ){
   if(coid+1 == csize)
     realloc(...);
   bw[coid++] = t.rid;
 }
}
```

```
for(t in T.records){
 if( pred(t) ){
   parent.consume(t);
   if(coid+1 == csize)
     realloc(...);
   bw[coid++] = t.rid;
 }
}
```

Figure 6.11: Provenance capture on selection under INJECT and DEFER semantics.

Consider the examples in Figure 6.11 that perform provenance capture on the selection operator under DEFER (left) and INJECT (right) scheduling of the instrumentation logic. The source code for the inject provenance capture in Figure 6.11(right) is the same with the one we generated in Section 6.4.2 by implementing the `on_after_parent` function of the selection instrumentor (modulo the initialization steps for better presentation). The defer approach in Figure 6.11(left) first executes the selection. Then, it re-executes the selection, this time without calling the parent, and applying the logic of `on_after_parent`.

The way that SMOKE defers the instrumentation logic for instrumentors share many similarities across operators. In fact, a natural classification for the purposes of deferring instrumentation logic is whether an operator is stateful or stateless. If the operation is stateful, SMOKE will attempt to reuse the state for the purposes of deferring the instrumentation logic. If the operator is stateless SMOKE will re-execute the operator. In both cases, SMOKE needs to guarantee several invariants so that the deferred instrumentation logic is guaranteed to result in the same outcome as the injected one. Next, we discuss how SMOKE defers the instrumentation on selections and hash-based group-by aggregations to illustrate differences between deferring instrumentation logic on stateless and stateful operators.

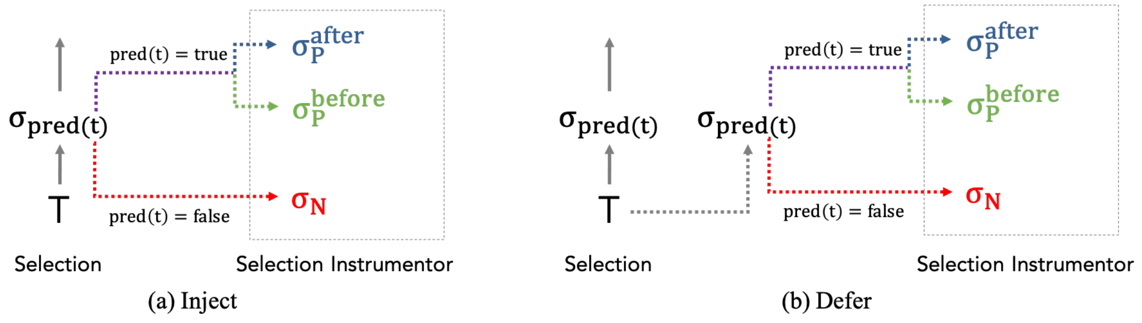
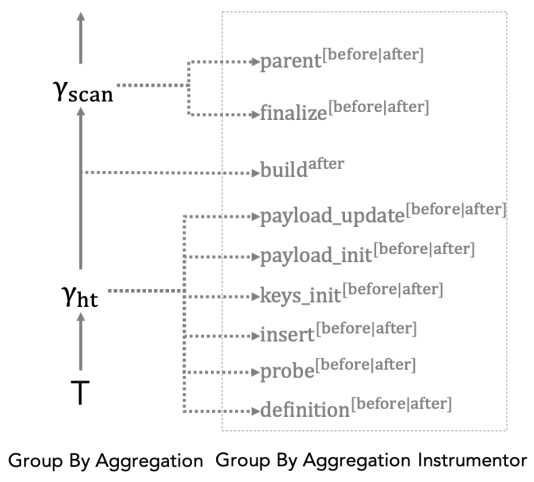
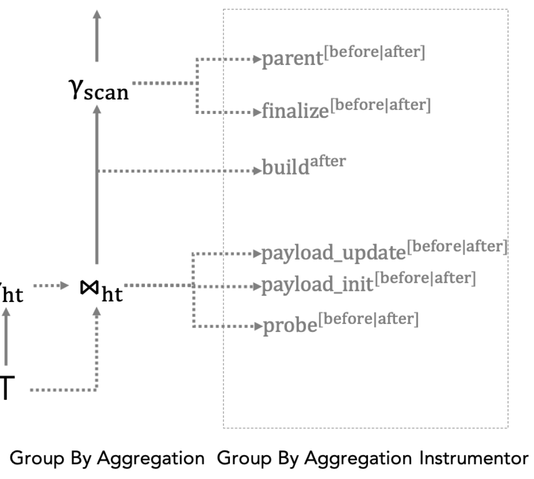


Figure 6.12: Deferred instrumentation on selections.

**Selection.** In the general case, SMOKE defers the instrumentation logic on a selection by simply re-executing the selection and applying the instrumentors $\sigma_P^{\texttt{before}}$; $\sigma_P^{\texttt{after}}$; and $\sigma_N$ without calling the parent of the selection. The problem with this approach is that if the instrumentors depend their logic on the selection's parent consumption, then the parent needs to be re-executed as well. To account for this semantics, SMOKE will execute the parents of the selection as long as the parents do not change the state of the database and the state of the clients that consume the results of a query (i.e., the parent consumption is pure). For instance, if the query performs an update and the selection instrumentor depends on this update, then SMOKE will not perform the instrumentation on a deferred fashion. However, since SMOKE targets analytical workloads, our focus is on queries that only perform reads or creates views. As such, there are two cases where re-executing parents is problematic: when the final operator performs a projection to send tuples to the client or when the final operator performs a view materialization. In both cases, SMOKE re-executes the parent of the selection but alters the final projection and materialization operators with dummy ones (i.e., a projection that sends the results to a dummy client and a materialization operator that creates a temporary view that will be purged upon completion of the deferred logic).



(a) Inject



(b) Defer

Figure 6.13: Inject (left) and defer (right) instrumentation on group-by aggregations.

**Hash-based group-by aggregations.** Similarly to selections, instrumentors on hash-based group aggregations can be deferred by re-execution. The main problem with this naive approach is that the hash table constructed for grouping purposes will need to be reconstructed—which is an expensive operation. To account for this problem, SMOKE pins the hash table constructed during normal query execution. Then, it reuses it for regenerating the data flows for instrumentation points. This is accomplished by the operator $\bowtie_{\mathrm{ht}}$ (see Figure 6.13(right)) that probes the pinned hash table with the input of the group-by aggregation. As an example, this is exactly how we implemented the DEFER provenance capture of group-by aggregations in Section 3.4.3. Note, however, that this technique does not support deferring the logic of every instrumentation point of the group-by aggregation. In particular, deferring the instrumentation points on hash table definition is meaningless since altering the hash table definition cannot happen in deferred execution (i.e., the hash table is already used by the group-by aggregation). Furthermore, deferring the initialization of keys and insertions of entries in the hash tables are also meaningless since the hash table is already built post-execution. Now, as shown in Figure 6.13(right), what SMOKE supports is the automatic defer of the payload initializations and payload updates but only for attributes that are calculated by instrumentors (e.g., the backward rid arrays for provenance capture). SMOKE also supports deferring the probing points. However, it should be noted that probing at defer mode is different from probing during the group-by aggregation. This is because in defer mode the hash table is already constructed. This is an important note for monitoring applications that may want timing statistics of the hash table probing phase.

### 6.5.2 Manual Defer

So far, we have introduced cases when SMOKE defers the instrumentation logic automatically. However, instrumentors can also defer their instrumentation logic on their own. Recall from our discussion in Section 6.3 and our descriptions in Table 6.1 that plans and pipelines provide the instrumentation points $\mathrm{P}^{\mathrm{after}}_{\mathtt{end}}$ and $\vdash^{\mathtt{after}}_{\mathtt{end}}$ on their end of their execution. These points are essentially the spots where the defer logic for an operator can be inserted. Hence, by implementing the instrumentation functions associated with these points, applications can provide a manual implementation of their deferred logic. In fact, this is also how SMOKE implements the automatic defer functionality that we discussed above.

### 6.5.3 Partial Inject-Partial Defer

Both manual and automatic defer are considered with the case where the whole instrumentation logic is deferred. In practice, as we showed with provenance capture (Chapter 3) and provenance analytics (Chapter 4), applications typically inject some parts of the logic and defer the rest. In this direction, applications can use the defer in connection with the inject scheduling capabilities of SMOKE to implement partial inject-partial defer schemes.

The main design principle behind how to develop such partial inject and partial defer techniques is the following. First, injected instrumentation logic creates some state either by piggybacking it into data structures created during query execution or by maintaining it in storage handled by instrumentors. Then, the deferred instrumentation logic reads the state generated by the injected instrumentation logic, the state generated by the plan, and the state generated by previous pipelines to implement the remaining instrumentation logic.

This is a very important design principle for the implementation of instrumentors that we believe contributes to best practices when designing applications. Furthermore, it demonstrates optimizations opportunities that future optimizers of instrumented plans should consider. For instance, recall the declarative specification of instrumentors. If a query that expresses the implementation logic is complex enough future optimizers of instrumented plans should automatically decide which portions of the query to inject and which to defer.

### 6.5.4 Execution Orders

Instrumentors that implement the same instrumentation function should be ordered with regards to whose instrumentation function will be executed first. As we discussed in Section 6.4.5, instrumentors need to register themselves to operators so that they can be called when instrumentation points are reached. A natural way to order instrumentors of the same instrumentation points is by the order they register themselves in the operator. In the general case, this ordering works unless an instrumentor A depends its instrumentation logic on the outcome of an instrumentor B that is ordered after A. For such cases, instrumentors can also provide the order that SMOKE should execute the registered instrumentors. This can be accomplished by specifying an order number in the register functions of Section 6.4.5.

In this section, we introduced the Scheduler component of the Physical Plan Instrumentation Framework of SMOKE. Next, we continue with our description of the components of the Physical Plan Instrumentation Framework by introducing the Storage Manager.

## 6.6 Storage Manager

Instrumentation applications across domains need access to storage for three purposes. First, to manipulate the state of operators (e.g., the hash table of a hash join) based on their instrumentation logic. Second, to implement their instrumentation logic or to keep a state post-instrumentation to allow their clients to query over it. Third, to access and alter the physical database design of the database during query execution (e.g., as is the case for adaptive physical database design techniques such as database cracking).

To account for this functionality, the Storage Manager of the Physical Plan Instrumentation Framework provides applications with functions to read, write, and modify the state of physical operators as well as create, read, write, and modify their internal storage or the storage of the database within the implementation of instrumentation functions.

**Challenges.** There are two main technical challenges related to the Storage Manager that we address in Sections 6.6.1 and 6.6.2. The first one regards the programmatic interface for creating, reading, writing, and modifying either the internal storage of SMOKE, the state of operators, or the internal state of instrumentors. The second challenge regards the operator state access. Modification of the internal state of an operator means that the semantics of the initial physical plan may be violated and applications may need a guarantee that the instrumented physical plan will produce the same results with the initial physical plan. For instance, a common case, as we showed in provenance capture, is to augment hash tables of hash joins or hash-based group-by aggregations with rid arrays so that instrumentors of parent operators can use this information for provenance capture purposes. In such cases, the instrumented physical plan still has to produce the same output with the initial plan no matter the changes in the state of intermediate operators.

**Solutions overview.** To address the first challenge, SMOKE exposes its internal type system out of which instrumentors can define complicated data structures. To operate on data structures, SMOKE exposes programmatic APIs over them. To address the second challenge, SMOKE introduces techniques that guarantee the semantics of initial plans when applications access the internal state of operators. In SMOKE, such states can be either hash tables or relations, and our discussion focuses on them.

Next, we first provide an overview of the type system and data structures of SMOKE that applications can use for accessing the storage of SMOKE (Section 6.6.1)—to address the first challenge. Then, we discuss how SMOKE guarantees the semantics of initial plans when instrumentors change the operators' state (Section 6.6.2)—to address the second challenge.

### 6.6.1 Access to SMOKE's Storage

As we noted in Section 6.4, instrumentors can be treated as new operators that applications introduce in a plan. Hence, similarly to how physical operators in SMOKE use its underlying type system and data structures to implement their logic so is the case for instrumentors. Next, we provide an overview of the type system and data structures that SMOKE provides.

#### 6.6.1.1 Type system

The primitive types that SMOKE provides follow the ones exposed by every major database. More specifically, SMOKE provides `INTEGER` (32-bit signed), `BIGINT/LONG` (64-bit signed), `REAL/FLOAT` (IEEE 754 binary32), fixedpoint `DECIMAL` types, fixed-length `CHAR` strings, variable-length `VARCHAR` strings, `DATETIME/TIMESTAMP` (ms resolution), `BOOLEAN` (1 byte; or 1 bit if NULLs are not allowed), and `CBLOB/BLOB` types. Furthermore, SMOKE provides `RID` types for row identifiers (rids) at different resolutions (i.e., 4, 8, 32, and 64-bit unsigned). Note that types can be used by both interpretation- and compilation-based instrumentors that we discussed. In interpretation-based instrumentors they can be used directly as C++ types. In compilation-based ones, SMOKE's compiler exposes these primitive types and associated operations on them (e.g., addition, subtraction, multiplication, and division) in its internal IR and compiles them into their equivalent C++ types. As an example, consider the `RID coid` that we introduced for provenance capture purposes in the interpretation-based instrumentor. In the compilation-based one, we used `RIDVariable` instead, which is the RID type as provided in the internal IR of SMOKE. At compilation time, the `RIDVariable coid` will be compiled into `RID coid` by the Compiler of SMOKE.

#### 6.6.1.2 Data structures

Out of primitive types now, instrumentors can compose complex data structures. SMOKE exposes records (each corresponding to a `struct` with fields typed based on primitive types or other data stuctures), arrays (single or multidimensional), hash tables, relations (implemented as arrays of records), and (dense or sparse) graphs. Note that SMOKE is an in-memory database engine. Hence, all data structures have in-memory representations. The important thing with SMOKE is that all data structures can be defined in its internal IRs. Hence, compilation-based instrumentors can use them to express their logic and, in turn, SMOKE will compile them into specialized data structures. This is important for both performance optimization as well as for ease of expressing the instrumentation logic.

As an example, consider again the specialized provenance indexes that we introduced in Chapter 3. These indexes are constructed by specializing the graph data structure (i.e., an inverted list) that SMOKE provides so that each entry maintains an RID list. Similarly, using the graph data structure of SMOKE we can use it to maintain lists of records (each record is a `struct` with typed fields) instead of RIDs. This is how we constructed the representation for the group-by aggregation push down optimization in Chapter 5. Hence, out of a single data structure specified in the internal IR we can devise highly-performant data structures specialized for our scenario at hand. This demonstrates the power of compilation of data structures that instrumentors can also use for their specialized instrumentation logic.

Furthermore, on top of data structures SMOKE also provides programmatic APIs for writing and reading them. Since the data structures that SMOKE provides (i.e., hash tables, arrays, relations, and graphs) have well-known APIs we omit further discussion.

On a final note, our discussion above focuses only on how instrumentors can define and use data structures for their own logic. However, the same data structures are used by

SMOKE to store relations, views, and indexes. As such, instrumentors can use the same APIs to read and write the underlying physical database design.

### 6.6.2 Operator State Access

In this section, we discuss how SMOKE enables instrumentors to write intermediate relations and hash tables, which are the two basic forms of state maintained in plans, all while guaranteeing the semantics of initial physical plans before instrumentation. More specifically, we discuss how SMOKE guarantees semantics if instrumentors add or remove keys and payload attributes in hash tables. In our discussion, we focus on guaranteeing the semantics of the hash-based group-by aggregation to highlight the main concepts behind our approaches. (Similar are our approaches for other operators that maintain hash tables).

#### 6.6.2.1 Adding keys

An instrumentor may add more attributes to keys of the hash-table maintained by a group-by aggregation typically because it needs to piggyback computation in the current hash table construction while the parent operators do not change and still require to consume the initial result. As an example, an instrumentor may add keys in the hash table because it needs to pre-compute drill down aggregates. SMOKE guarantees the semantics of the initial group-by aggregation with the following approach:

Recall that a group-by aggregation is implemented by building the hash table ($\gamma_{\mathrm{ht}}$) and then scanning it (i.e., $\gamma_{\mathrm{scan}}$) to finalize aggregates. To guarantee the semantics of the initial group-by aggregation, $\gamma_{\mathrm{ht}}$ performs grouping on the union of the initial keys with the keys of the instrumentors. At this stage, instrumentors can consume the result of grouping on the union of the keys. Before calling $\gamma_{\mathrm{scan}}$, $\gamma_{\mathrm{ht}}$ is followed by another operator $\gamma_{\mathrm{ht}}^{\mathtt{ROLLUP}}$ that rolls up the aggregates based on the initial keys. The result of the roll-up is essentially the result of $\gamma_{\mathrm{ht}}$ without instrumentation that $\gamma_{\mathrm{scan}}$ can consume. Note that to perform the grouping on the union of the initial keys with the keys of the instrumentors, instrumentors can either provide a new hash function (in Section 6.8.1 we discuss how to introduce hash functions in SMOKE) or default on the behavior of SMOKE that uses multiplicative hashing. Finally, note that if instrumentors have completely changed the plan (i.e., using the Actions component that we introduce in Section 6.8) and guaranteeing the semantics of the initial plan is not required, then $\gamma_{\mathrm{ht}}^{\mathtt{ROLLUP}}$ need not be called. Here, however, our discussion focuses on guaranteeing the semantics of the initial plan.

#### 6.6.2.2 Removing keys

Similarly to adding keys, removing keys may happen as a result of plan change (e.g., an online optimizer has determined a functional dependency between group-by attributes in which case dependent attributes can be removed from keys) or because the instrumentor wants to piggyback a computation in the current query (e.g., a roll-up computation). Again if the instrumentor does not remove keys due to plan changes we need to guarantee that parent operators consume the result of the initial $\gamma_{\mathrm{ht}}$. To guarantee the semantics of the initial group-by aggregation, we first perform $\gamma_{\mathrm{ht}}$ as requested by the initial plan. The result of $\gamma_{\mathrm{ht}}$ is consumed can be consumed directly by $\gamma_{\mathrm{scan}}$ per normal execution. This guarantees the semantics of the initial plan. To support instrumentors, however, $\gamma_{\mathrm{ht}}$ is also followed by another operator $\gamma_{\mathrm{ht}}^{\mathtt{ROLLUP}}$ that performs a rollup to group together on keys after the removal of the requested keys and combine together aggregates.

#### 6.6.2.3 Adding and Removing Payload Attributes

Similarly to adding or removing keys we can also add and remove attributes from payloads of a hash table. Adding attributes to payloads follows the logic of user-defined aggregates (UDAs). Instrumentors need to initialize, update, and finalize payload attributes. These operations can be defined by instrumentors by implementing the instrumentation points that correspond to init_payload, update_payload, and finalize_payload that we introduced in Section 6.3. In contrast to UDAs, however, note that SMOKE allows instrumentors to add any type of data element (i.e., from simple scalars to data structures such as graphs) as a payload attribute. For instance, in Chapter 5, we introduced a group-by aggregation push down optimization that pushes a hash table within each group computed by the group-by aggregation. This construction led us to derive a data cube.

To conclude, we note that techniques to handle the addition and removal of payload attributes are simpler than adding or removing keys because now we do not need to re-group or un-group due to key additions or removals, respectively. Payload attributes are simply propagated to instrumentors and not considered by parent operators.

#### 6.6.2.4 Access to intermediate relations

Finally, access to intermediate relations follows the logic of adding and removing payload attributes in hash tables. More specifically, instrumentors can add or remove attributes in relations that can be propagated to instrumentors and not considered by parent operators.

In this section, we introduced the Storage Manager component of the Physical Plan Instrumentation Framework of SMOKE. Next, we continue with our description of the components of the Physical Plan Instrumentation Framework by introducing the Announcer.

## 6.7 Announcer

The Announcer component of the Physical Plan Instrumentation Framework provides applications with the ability to specify runtime events (e.g., a hash table uses memory above a specified threshold, or the execution of an operator has exceeded a time threshold), get notified when such events take place, and react on these events in application-specific ways.

To enable this functionality in a principled manner, the Announcer component provides a function `on(Condition, Resolve, RequiredParemeters)` that applications can use to register a `Condition`, the function `Resolve` that will be executed when the condition is met, and the parameters `RequiredParemeters` to be passed to the `Resolve` function.

We illustrate the functionality provided with the `on` function with two examples: compression on provenance capture and online optimizer.

**Example 9 (Compression on Provenance Capture)** In our example of provenance capture on the selection operator in Section 6.4.2, we created an RID array `RIDArrayVariable bw`. A condition on the size of this array could be `bw.size > 30`. The RID array starts with 10 elements and is increased by a factor of 1.5 on reallocation. Hence, the condition `bw.size > 30` will be meet on the third reallocation call which will trigger a `Resolve` function. An interesting `Resolve` function, in this case, could be a compressor of the RID array. Finally, the `RequiredParameters` include the rid array variable `bw` and, possibly, the variables `coid` and `csize` in case compression decreases the size of the RID array. (This is because the variables `coid` and `csize` that keep where to place the next element and the current size, respectively, may need to be updated.) Hence, the `on` function can be specified as `on(bw.size > 30, compressor, {bw, coid, csize})`.

**Example 10 (Online Optimizer)** Similarly, the `Online Optimizer` example that we showed in Section 6.2.2 tracks the join cardinality online. The condition that the optimizer specifies is if the join cardinality goes above a threshold. We can express this condition as `join_cardinality > thr`. When this condition is met the `Resolve` function replaces the nested loop join with a hash-based one. (This replace operation is provided by the Actions component that we discuss in Section 6.8). Finally, the `RequiredParameters` include the nested loop and hash-based join physical operators. Hence, the `on` function can be specified as `on(join_cardinality > thr, replace, {NL,HJ})`.

Now, there are two main technical challenges behind the evaluation of `on` functions: how to decide when to evaluate a condition and how often to evaluate a condition. The first regards the fact that conditions need to be re-evaluated when variables involved in their specification (e.g., `bw` and `join_cardinality` in our examples) are updated during query execution. To actually trigger a re-evaluation, however, SMOKE needs to track when these variables are updated. The second regards the fact that evaluating a condition every time a variable involved in a condition is updated may add substantial overhead to query execution. Hence, applications need to specify how often conditions should be evaluated.

To address the first challenge, SMOKE performs static analysis of the instrumentation logic to derive what variables are involved in a condition. Then, during execution, updates on such variables trigger the re-evaluation of conditions. Currently, SMOKE supports only static analysis of its internal IR. Hence, this functionality is only supported for compilation-based instrumentors. To address the second challenge, SMOKE allows applications to specify how often a condition is evaluated in the `on` function. More specifically, the `on` function is extended to include an optional parameter, namely, `every`, that applications can set to a time interval passed which a condition needs to be evaluated. We believe more complicated schemes (e.g., trigger re-evaluation only after variables in the specification of a condition have been updated a certain number of times) are interesting future work.

In this section, we described the mechanisms provides by the Announcer component of the Physical Plan Instrumentation Framework so that applications can specify runtime events, get notified when such events arise at runtime, and react to these events in application-specific ways. Next, we conclude our introduction of the components of the Physical Plan Instrumentation Framework by introducing the Actions component.

## 6.8 Actions

We conclude our introduction of the components of the Physical Plan Instrumentation Framework of SMOKE by introducing the operations provided by the Actions component for changing plans structurally. More precisely, the operations that Actions provide include replacing, adding new, or removing physical operators from a plan as well as modifying operators by means of either changing their internal logic (e.g., adding or removing predicates from a selection) or their input and output schemas (i.e., by adding or removing attributes).

Providing these operations to applications, however, comes with multiple technical challenges that we address in this section. Next, we describe and address challenges behind changing schemas (Section 6.8.1), changing the logic of operators (Section 6.8.2) as well as replacing (Section 6.8.3), adding (Section 6.8.4), and removing (Section 6.8.5) operators.

## 6.8.1 Changing Input and Output Schemas

Each physical operator in a physical plan can have one or more inputs and an output that are defined from the optimizer as part of the generation of the physical plan. Instrumentation applications such as provenance managers or interactive data profiling frontends need to change these schemas to produce more or fewer attributes or piggyback computations within inputs and outputs to implement their instrumentation logic.

**Challenges.** There are two challenges involving changing input and output schemas of an operator. The first challenge is that applications need APIs to express what attributes to add to or remove from such schemas. The second challenge regards the fact that changing the output schema of an operator (i.e., to add or remove attributes) has an effect both on parents of the operator, because they now need to account for their changed input schema, as well as on its children, because they may need to produce more or fewer attributes.

**Solutions Overview.** To address the first challenge, SMOKE provides applications with APIs that alter the definition of input and output schemas (i.e., add or remove attributes from relations or keys and payloads from hash tables). To address the second challenge, SMOKE provides both automated methods to propagate schema changes to parent (i.e., to consume other inputs than the initial ones) and children operators (i.e., to produce more or fewer attributes for their parents) as well as disambiguation APIs for cases when propagating changes to parent and children is ambiguous and cannot be automated.

### 6.8.1.1 Schema Changing APIs

To better understand how instrumentors can alter input and output schemas, recall from Chapter 2 that each operator in SMOKE maintains a description of its inputs and output. For convenience, we replicate the interface of physical operator description `PhysicalOpPNodeDescription` from Chapter 2 in Figure 6.14.

A dataset description `DatasetDescription` is an object that defines the schema of inputs and outputs of physical operators. In SMOKE, there are two such types of `DatasetDescription` depending on the physical representation of the dataset: 1) a

```
struct PhysicalOpPNodeDescription{
 StateDescription state;
 bit is_blocking;
 DatasetDescription output;
 DatasetDescription left;
 DatasetDescription right;
};
```

Figure 6.14: Interface of physical operator description `PhysicalOpPNodeDescription`.

`RelationDefinition` and 2) a `HashTableDefinition` corresponding to cases where inputs or outputs are relations and hash tables, respectively. SMOKE extends the interfaces of `RelationDefinition` and `HashTableDefinition` to include functions that instrumentors can use to alter the schema of the corresponding relation and hash table.

Next, we first describe the APIs exposed from `RelationDefinition` and `HashTableDefinition`. Then, we discuss how SMOKE propagates changes on input and output schemas to parent and child physical operators.

### Relation Definition

```
struct RelationDefinition{
 Schema s;
 void add_attribute(Attribute);
 void remove_attribute(string);
};
```

Figure 6.15: Interface of relation definition `RelationDefinition` with functions for adding and removing attributes from its schema.

Consider the `RelationDefinition` interface in Figure 6.15. The interface maintains the schema `Schema` of the relation (i.e., attributes and their order in the relation) and exposes two low-level functions: `add_attribute` and `remove_attribute` that allow applications to append and remove attributes in and from the schema.

### Hash Table Definition

```
struct HashTableDefinition{
  KeysDefinition keys;
  PayloadDefinition payload;
  void add_key(KeyDefinition);
  void add_payload_attribute(Attribute);
  void remove_key(KeyDefinition);
  void remove_payload_attribute(Attribute);
  void set_hash_func(HashFuncDefinition);
  void set_equals(EqualsDefinition);
};
```

Figure 6.16: Interface of hash table definition `HashTableDefinition` with functions for adding and removing keys and payload attributes for its schema.

Consider the `HashTableDefinition` interface in Figure 6.16. The interface maintains they key and payload definitions and exposes 6 low-level functions (i.e., `add_key`, `remove_key`, `add_payload_attribute`, `remove_payload_attribute`, `set_hash_func`, and `set_equals`) that allow applications to alter the definition of keys, payload attributes, and internal `hash` and `equals` functions of the hash table. While each function name in `HashTableDefinition` is self-explanatory, we next provide their explanation for completeness and to introduce terminology that we use in subsequent sections.

**Adding and removing keys.** `add_key` and `remove_key` take as input a `KeyDefinition` and add it in or remove it from the `KeysDefinition keys`, respectively. A `KeyDefinition` is defined as a `struct` with a `name`, a `type`, and possibly an associated hash functionc `hfunc`. Finally, `KeysDefinition` maintains a map from the name of a key definition to the corresponding `KeyDefinition` object and the order of keys.

**Adding and removing payload attributes.** Similarly, `add_payload_attribute` and `remove_payload_attribute` take as input an `Attribute` and add it in or remove it from the `PayloadDefinition payload`. An `Attribute` is defined as a `struct` with a `name` and a `type`. (This is the same with attributes of a relation). Finally, `PayloadDefinition payload` maintains a map from attribute names to attributes and their order within the payload.

**Setting hash and equals functions.** Adding or removing keys results in a new key type for the hash table. Unless the resulting key type has a standard `hash` and `equals` function that SMOKE will pick automatically, application need to register a new `hash` and `equals` functions that SMOKE will use to hash keys and check for equality among them, respectively. This can be done through the function `set_hash_func(HashFuncDefinition)` and `set_equals(EqualsDefinition)` function of the `HashTableDefinition` interface. For interpretation-based instrumentors both `HashFuncDefinition` and `EqualsDefinition` are pointers to functions that perform the hash computation and check for equality among keys, respectively. For compilation-based instrumentors, `HashFuncDefinition` and `EqualsDefinition` are pointers to functions that define the `hash` and `equals` function in SMOKE's internal IR.

Having described the two main forms of datasets, we next discuss how changing schemas forces SMOKE to automatically propagate changes to parent and child operators and cases when applications need to introduce their own logic for changing parent and child operators.

```
O = SELECT   year, product,
             COUNT(*) AS cnt,
             SUM(revenue) AS total_revenue
    FROM     SALES
    GROUP BY year, product;
```

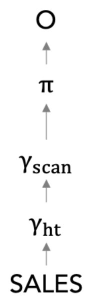


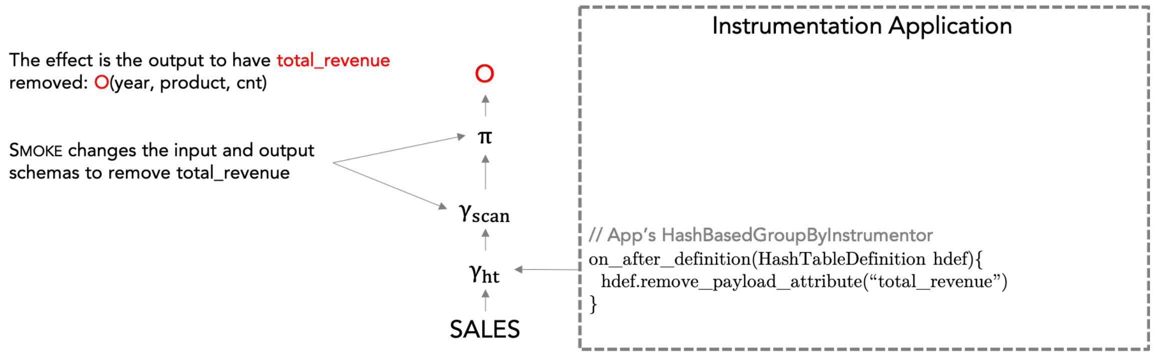
Figure 6.17: Example group by query and corresponding physical plan that we use in our discussion.

To better explain the different techniques, we use a group by aggregation example and corresponding physical plan (see Figure 6.17) that computes the number of sales and overall revenue per year and product over a sales table `SALES(revenue,year,product)`.

Furthermore, we limit the discussion around changing keys and payload attributes of hash tables. (Similar are our techniques for changing relations and we omit a discussion on them.)

#### 6.8.1.2 Notifying Parents

If an instrumentor decides on altering the keys and payload attributes of a hash table maintained by an operator, then parent operators may no longer be able to operate given that their initial input schema (i.e., the schema of the hash table or relation) has changed. In Section 6.6.2, we covered cases when parent operators need to continue to operate on the outputs of the initial operators. Here, we discuss how to propagate changes (i.e., parent operators change as an effect of changing the schemas of their children).

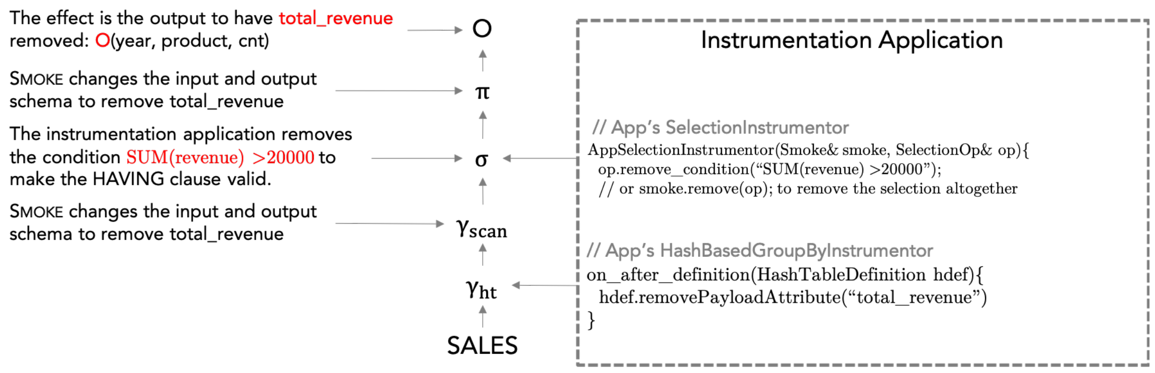

The effect is the output to have total_revenue removed: O(year, product, cnt)
SMOKE changes the input and output schemas to remove total_revenue
O
π
γscan
γht
SALES
Instrumentation Application

```
// App's HashBasedGroupByInstrumentor
on_after_definition(HashTableDefinition hdef){
  hdef.remove_payload_attribute("total_revenue")
}
```

Figure 6.18: Propagation of changes to parents upon removing a payload attribute (i.e., `SUM(revenue)`) from the hash table maintained by $\gamma_{\text{ht}}$ of our group-by aggregation example.

**Removing keys or payload attributes**

Upon removal of a key or payload attribute, SMOKE will propagate the removals to parent operators, if necessary. Consider again our example in Figure 6.17. If an instrumentor of $\gamma_{\text{ht}}$ removes the payload attribute that corresponds to `SUM(revenue)`, SMOKE will remove from the result `O` the attribute `SUM(revenue)`. To do so, SMOKE removes the corresponding `SUM(revenue)` attribute from the schema definitions of the operators $\gamma_{\text{scan}}$ and $\pi$. For completeness, the overall process is illustrated in Figure 6.18.

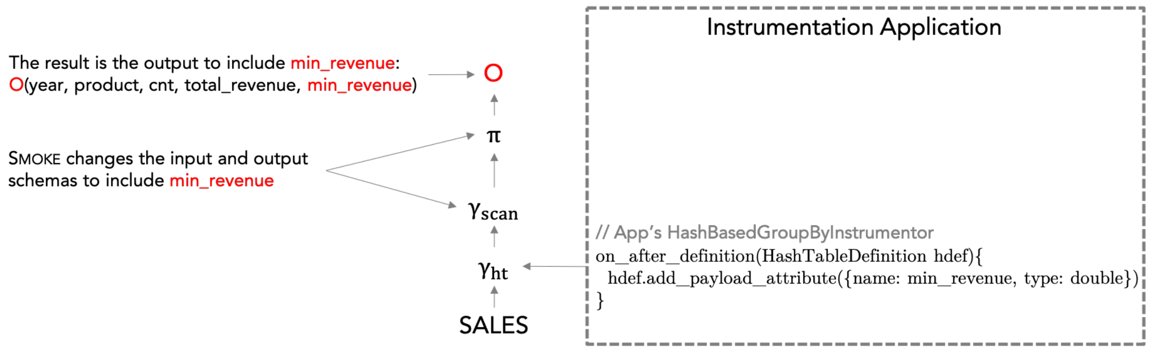

The effect is the output to have total_revenue removed: O(year, product, cnt)
Smoke changes the input and output schema to remove total_revenue
The instrumentation application removes the condition SUM(revenue) >20000 to make the HAVING clause valid.
Smoke changes the input and output schema to remove total_revenue
O
π
σ
γscan
γht
SALES
Instrumentation Application

```
// App's SelectionInstrumentor
AppSelectionInstrumentor(Smoke& smoke, SelectionOp& op){
  op.remove_condition("SUM(revenue) >20000");
  // or smoke.remove(op); to remove the selection altogether

// App's HashBasedGroupByInstrumentor
on_after_definition(HashTableDefinition hdef){
  hdef.removePayloadAttribute("total_revenue")
}
```

Figure 6.19: Propagation of changes to parents upon removing a payload attribute (i.e., `SUM(revenue)`) and resolution of ambiguities by instrumentors.

Now, if removed keys or payload attributes are involved in the logic of parent operators, then Smoke will not provide a resolution on its own but rather expects the application to introduce further instrumentors to provide a resolution. Smoke does so because the resolution logic depends on the application logic. To illustrate, consider a variant of the query `O` that also has a having clause `HAVING SUM(revenue)>20000 AND COUNT(*) < 15000`. Removing `SUM(revenue)` from $\gamma_{ht}$ invalidates the `HAVING` clause. For its resolution, the instrumentation application could remove the `HAVING` clause altogether or only remove the clause `SUM(revenue)>20000`. Both operations are provided by actions on plans that we introduce in Section 6.8. However, deciding how to resolve the inconsistency is application-dependent. For completeness, the process is illustrated in Figure 6.19.

Finally, note that the removal techniques that we have shown so far can be applied both at compile time as well as during execution. The difference is that during execution Smoke will also purge potential materialization of removed attributes. Furthermore, while our discussion focuses on removing keys and payload attributes from hash tables the same propagation techniques apply for removing attributes from relations.

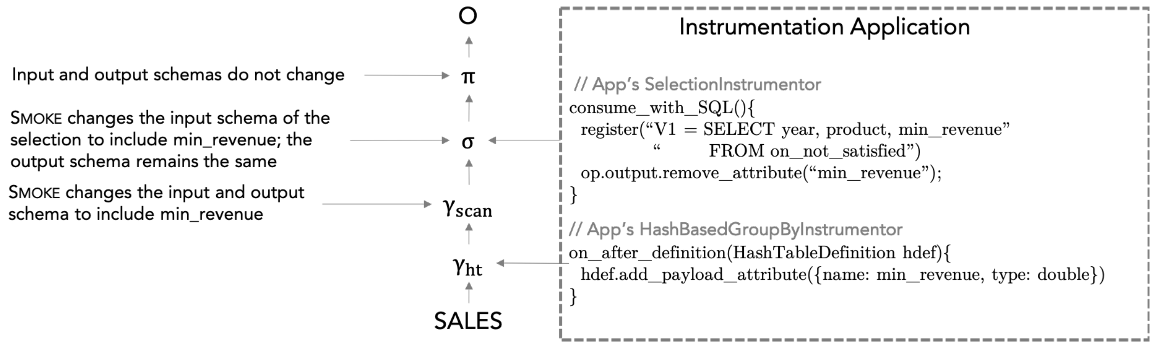

Instrumentation Application
The result is the output to include min_revenue:
O(year, product, cnt, total_revenue, min_revenue)
O
π
SMOKE changes the input and output schemas to include min_revenue
$\gamma_{scan}$
$\gamma_{ht}$
SALES

```
// App's HashBasedGroupByInstrumentor
on_after_definition(HashTableDefinition hdef){
  hdef.add_payload_attribute({name: min_revenue, type: double})
}
```

Figure 6.20: Propagation of changes to parents upon adding a payload attribute (i.e., `MIN(revenue)`) to the hash table maintained by $\gamma_{\text{ht}}$ of our group-by aggregation example.

## Adding keys or payload attributes

Upon addition of a key or payload attribute, SMOKE will propagate the additions to parent operators similarly to how it propagates changes upon their removal. For instance, suppose that we want to compute the `MIN(revenue)` along with the other aggregates of `O`. In this case, an instrumentor should add another attribute to the hash table maintained by $\gamma_{\text{ht}}$. In turn, SMOKE will propagate the addition to parent operators as shown in Figure 6.20.


Instrumentation Application
O
Input and output schemas do not change
π
SMOKE changes the input schema of the selection to include min_revenue; the output schema remains the same
σ
SMOKE changes the input and output schema to include min_revenue
$\gamma_{scan}$
$\gamma_{ht}$
SALES

```
// App's SelectionInstrumentor
consume_with_SQL(){
  register("V1 = SELECT year, product, min_revenue"
           "        FROM on_not_satisfied")
  op.output.remove_attribute("min_revenue");
}
// App's HashBasedGroupByInstrumentor
on_after_definition(HashTableDefinition hdef){
  hdef.add_payload_attribute({name: min_revenue, type: double})
}
```

Figure 6.21: Example of propagating changes to parents upon adding a payload attribute in a hash table.

Instrumentors, however, may also want to add keys and payload attributes so that only instrumentors of parent operators can consume the added keys or payload attributes. For instance, consider again our example with the added `HAVING` clause and adding `min_revenue` in the payload attributes. Now, suppose that we do not want `min_revenue` to be propagated to the output. Rather, we want to make an instrumentor of the `HAVING` clause that stores the `min_revenue` for every `year, product` group that was filtered out (e.g., for data explanation purposes). Figure 6.21 shows this process. To stop the propagation of the `min_revenue` to the parent of `HAVING`, instrumentors simply need to remove the attribute from the output schema of the `HAVING` operator. In turn, SMOKE will stop propagating the addition to parent operators.

Now, note that in contrast to removing keys and payload attributes, adding keys or payload attributes does not conflict with logic introduced in parent operators. Hence, instrumentors do not need to resolve ambiguities. Furthermore, similarly to removing keys and payload attributes, the techniques are similar for propagating changes upon adding attributes to relations, and we omit further discussion. Also, the propagation techniques upon adding keys or payload attributes can be applied at either compile or run time. In case the addition happens at runtime, SMOKE needs to extend relations and hash tables with more space to keep the new attributes. Finally, adding keys complicates the logic of the operator that maintains the hash table, and SMOKE provides automated ways for its resolution that we presented in Section 6.6.2.

#### 6.8.1.3 Notifying Children

Similarly to how we notified parent operators for changing the schema of a hash table, we also need to update children to a) propagate more attributes, if needed, due to the addition of payloads attributes and b) do not propagate attributes if these are only involved in keys and payload attributes that are removed. Both operations are handled by SMOKE automatically.

In the case of adding keys and payload attributes, instrumentors should request more attributes from base relations involved in the computation. In turn, SMOKE propagates down to scans of base or intermediate relations the request to produce more attributes as well as changes the input and output schemas of intermediate operations to include potential new attributes. Note that at the stage of adding new keys and payload attributes, however, the way an instrumentor wants to compute their value is not known yet (i.e., the values of keys and payload attributes are computed based on the instrumentation logic). Hence, SMOKE will propagate the request for attributes to children only when instrumentors specify the logic on how to compute a new value and associate it with a new key or payload attribute. This happens within instrumentation functions that we discussed in Section 6.4.

Similarly, when instrumentors remove keys and payload attributes, SMOKE will ask child operators to produce fewer attributes. Note that in contrast to adding keys, SMOKE will stop asking children not to produce attributes if these are involved in the logic of some child operation. To illustrate, consider that a modification to our example query `O` to include a selection on revenue (e.g., `revenue > 100`) before performing the group-by aggregation and an instrumentor requires that we remove `SUM(revenue)`. In this case, SMOKE will still require the initial scan operation to provide the revenue attribute so that the selection is still valid, the input schema to the selection will stay the same, but the output schema of the selection and the input to $\gamma_{\text{ht}}$ will not include the `revenue`. In contrast, if the selection on `revenue` was not present, as in our initial query `O`, then SMOKE asks the scan to not produce `revenue` and will change the input schema to $\gamma_{\text{ht}}$ to not consider it.

So far, we have described how SMOKE enables instrumentors to alter the definitions of input and output schemas of physical operators as well as how the new definitions can be propagated in parent and child operators. Next, we present techniques that allow instrumentation application to change the internal logic of operators.

### 6.8.2 Changing Internal Logic

Changing the input and output schemas of an operator is one way to change the internal logic of an operator. Another way involves changing how operators compute their internal logic. Next, we address how to change the internal logic of two important operators (i.e., selections and joins) through changing selection and join predicates.

```
struct CNF{
  Tree and_or;
  void remove_condition(Condition);
  void add_condition(Condition);
  void change_condition(Condition old_condition,
                        Condition new_condition);
  virtual void navigate();
};
```

Figure 6.22: Interface of `CNF` with functions for adding, removing, and changing conditions.

### Selection

The internal logic of a selection operator is defined by the selection predicates that it evaluates. In SMOKE a selection predicate is an AND-OR tree that encodes the CNF condition of the selection. SMOKE allows instrumentors to alter the AND-OR tree by either adding, removing, or changing conditions. To enable this functionality, the interface of CNF (see Figure 6.22) includes four functions (i.e., `add_condition`, `remove_condition`, `replace_condition`, and `navigate`) that instrumentors can use to add new conditions, remove current conditions, replace conditions, and navigate the AND-OR tree to add conditions in specific sub-conditions. Finally, note that SMOKE also accounts for cases where conditions repeat in a CNF. (For such cases, the AND-OR representation is a graph rather than a tree.) If applications want to remove or replace a condition that repeats in a CNF, they need to navigate the graph to remove or replace repeated conditions explicitly.

### Joins

Similarly to selections, joins are also defined based on join conditions that are represented as AND-OR CNFs. Hence, applications can alter the join predicate similarly to how they alter the selection predicate. An important distinction on joins is that applications are not allowed to violate the semantics of the operator. For instance, if the join operator is a hash-based one, applications are not allowed to add conditions that involve inequalities among the join keys. If an instrumentor wants to perform such an operation, it has to replace the hash-based join with an equivalent under replacement nested loop join, as we discuss next.

### 6.8.3 Replacing Physical Operators

**Challenges.** Replacing operators comes with two challenges: First, we need to guarantee the correctness of the initial plan under replacement. For instance, if we replace a nested loop join with a hash join during execution, the hash join needs to be set in such a way so that it does not produce the results that the nested loop join has already produced. Note, however, that this correctness is not always required. For instance, an application may want to replace the nested loop join with ripple [HH99] or wander [LWYZ16] join and applications should be able to specify if the semantics of the initial plan should be guaranteed. Second, replacing operators that are already instrumented means that instrumentors of the old operator will become invalidated and new instrumentors need to be introduced for the new operator.

**Solutions overview.** To address the first challenge, SMOKE introduces classes of operator equivalences under replacement (e.g., by introducing implementations of nested loop joins that can replace hash join implementations during execution) as well as allows instrumentation applications to replace physical operators with operators that define on their own. To address the second challenge, SMOKE allows applications to assign instrumentors of old operators to new ones. Furthermore, SMOKE can also assign old instrumentors to new ones automatically, but applications need to request such functionality. For instance, consider the `Monitoring` application in Figure 6.2 that we discussed in Section 6.2.2. Whether we change the nested loop join to a hash join, the logic of how to compute the time spent on parents does not change. As such, instrumentation applications can either assign the logic of the `before_parent` and `after_parent` of the nested loop join to the ones of the hash join or ask SMOKE to perform such assignments automatically. Note that in case that such assignments are not possible then instrumentation applications need to introduce new instrumentors for the new operator.

#### 6.8.3.1 Equivalence Under Replacement

To address the first challenge, we introduce the notion of *equivalence under replacement* as a property of two operators, say, $\mathrm{A}$ and $\mathrm{B}$, as follows:

**Property 1** *Two operators* $\mathrm{A}$ *and* $\mathrm{B}$ *are equivalent under replacement iff:*

1. $\mathrm{B}$ *does not produce any tuples that have already been produced by* $\mathrm{A}$
2. $\mathrm{B}$ *generates all tuples that would be generated by* $\mathrm{A}$*, if* $\mathrm{A}$ *was not replaced by* $\mathrm{B}$.

If two operators share this property then the following holds:

**Theorem 1** *Consider a physical plan* $\mathrm{P}$ *and a physical operator A in* $\mathrm{P}$*. Furthermore, consider that we want to replace* $\mathrm{A}$ *with an operator* $\mathrm{B}$ *in the plan. If* $\mathrm{A}$ *and* $\mathrm{B}$ *are equivalent under replacement then the new plan* $\mathrm{P}'$ *that has* $\mathrm{B}$ *in the place of* $\mathrm{A}$ *guarantees the semantics of the initial plan* $\mathrm{P}$.

**Proof 1** *The proof is straightforward. We show that by contradiction. Assume that replacing* $\mathrm{A}$ *with* $\mathrm{B}$ *does not guarantee the semantics for the query. Consider* $\mathrm{R(A)}$ *to denote the results of* $\mathrm{A}$*. Also, consider the execution of* $\mathrm{A}$ *up to a specific point. Denote this partial execution as* $\mathrm{R(A')}$*. Then the execution of* $\mathrm{B}$ *guarantees that it generates* $\mathrm{R(B)}$ *s.t.* $\mathrm{R(A')} \cup \mathrm{R(B)} = \mathrm{R(A)}$*. Producing* $\mathrm{R(A)}$*, however, guarantees the semantics of the query which leads us to a contradiction. Hence, replacing* $\mathrm{A}$ *with* $\mathrm{B}$ *guarantees the semantics of the physical plan.* $\square$

Hence, to address the first challenge our goal is to introduce techniques that given an operator $\mathrm{A}$ can generate an operator $\mathrm{B}$ that guarantees the two conditions of Property 1. For instance, if we are given a nested loop join at some point during its execution, our goal is to provide a hash join implementation that guarantees the two conditions of Property 1. For optimization purposes, however, we should also avoid introducing new operators and plans that perform again work already performed by the initial plan. For instance, if the nested loop join has computed some join results then the hash join should build on this work.

In Section 12.2, we will introduce operator implementations in such equivalence classes. Here, we note that SMOKE allows applications to replace operators by picking other operators from their equivalent under replacement classes or by introducing their own operators.

#### 6.8.3.2 Assigning Instrumentors

The address the second challenge, SMOKE allows applications to assign instrumentors of the old operator to the new ones. This can happen based on the assignment operator of instrumentation functions. For instance, a statement `NL.on_before_parent = HJ.on_before_parent` when an application replaces the nested loop (`NL`) join with a hash-based (HJ) one, forces SMOKE to execute the `on_before_parent` of the NL join when it executes the HJ one. SMOKE also maps semantically equivalent instrumentations points, such as `before_parent` of a nested loop join with the before_parent of a hash join. Applications can then ask SMOKE to map the instrumentation functions of the old operator to the new one automatically. We omit a description of these mappings since they are evident from our discussion in Section 6.3.

So far, we have covered how Actions enable applications to modify the schemas of operators, change their internal logic, and how to replace operators. Next, we conclude our discussion on Actions by briefly discussing how applications can add and remove operators.

### 6.8.4 Adding Physical Operators

Instumentation applications such as probabilistic predicates [LCKC18], lookahead [PDZ+18] and sideways [IT08] information passing, or techniques that introduce vectorization in compilation-based engines [MMP17] need to add operators within plans.

To accommodate this functionality, SMOKE's instrumentation framework allows the addition of a new operator in a plan with the function `add_operator(Operator new_op, Operator parent_op)`. This operation will add the operator `new_op` as a child of the parent `parent_op` in the plan. (Note that to add an operator as the new root operator of the plan, applications can specify the parent as NULL.) Furthermore, every operator added in a plan should be defined under the producer-consumer prototype. Based on the definition of added operators, SMOKE identifies what new attributes may be needed by child operators and propagates these requests as we discussed in Section 6.8.1.3. Finally, note that besides the `add_operator` function, applications are also allowed to directly add operators by manipulating the physical plan tree, as defined in Chapter 2.

There are two important notes concerning additions of operators; one on performance and another on correctness. We address them below:

First, when new operators are added at execution time, a compilation cost needs to be paid for generating the source code of the new plan. SMOKE can mitigate this cost at compile time if instrumentors specify their logic for adding operators before execution. In this case, SMOKE pre-compiles the plans with the added operators and switches between binaries (i.e., from the plan without the addition to the plan with the addition) at run time.

Second, adding operators should conform to the way the processing was happening before the addition. For instance, if we introduce an operator that takes as input records and batches them (e.g., as is performed by query-compiled engines to perform vectorization [MMP17]) then it is not just sufficient to add a batch operator. Parent operators should be changed as well. Essentially, this means that the input schema and its physical representation of the added operator should conform with the input schema of its parent while the output schema and physical representation of the output of the added operator should conform with the input of the parent operator. In any other case, a) other new operators should be added, so that parent operators can continue consuming and children can continue producing on the same way as before or b) operators of the initial plan should be changed (using other actions described so far) to account for the newly added operators.

### 6.8.5 Removing Physical Operators

Similarly to adding operators, SMOKE also allows applications to remove operators from a plan. There are two types of removal: removing a single operator and removing the whole subplan rooted at a specific operator. SMOKE handles both cases by exposing a function `remove_operator(Operator, [SINGLE|WHOLE])` that either removes the operator from the plan or removes the whole plan rooted at the operator.

There are four concerns regarding removing operators. First, removing individual operators in SMOKE results in connecting the child of the operator with its parent. Under this semantics, not every removal is possible. For instance, consider removing a join whose inputs were two base relations and its output is passed to a group-by aggregation. There is no semantically meaningful way of connecting two base relations with the group-by aggregation. For such cases, applications should follow up by removing other operators of the plan to make it valid. Second, removing the whole subplan rooted at an operator when the operator is not the root of a pipeline, results in the pipeline being left hanging. In such cases, SMOKE will remove all operators up to the root of the pipeline. If this operation is performed at runtime and the pipeline has produced some results, then next pipelines will proceed with the partial result as created by the so-far execution of the removed pipeline. The third and fourth concerns are the same with the concerns of adding operators in a pipeline (i.e., the compilation cost at runtime that can be mitigated to compile time if the removal logic is fixed at compile time and that the removal should preserve the invariant that the output of the child of the removed operator with the input of the parent of the removed operator should have matching schemas and physical representations).

With the introduction of the Actions component, we have concluded with the overall description of the Physical Plan Instrumentation Framework. Next, we discuss how we changed database components (i.e., compiler, physical algebra, and optimizer) in support of the Physical Plan Instrumentation Framework.

## 6.9 Changes on Database Components

We conclude our discussion on physical plan instrumentation by discussing changes that we had to make in underlying database components in support of instrumentation.

### 6.9.1 Instrumentation-Aware Compiler

SMOKE under normal query execution compiles physical plans to source code following the producer-consumer compilation model [NL14], as we discussed in Chapter 2. Under instrumentation, however, SMOKE needs to compile instrumented physical plans. To compile an instrumented physical plan, SMOKE also follows the producer-consumer compilation model [NL14]. The main idea is that instrumentation points can be considered as points in plans from where instrumentors *consume* from. Intuitively then, under the producer-consumer compilation model, instrumentors should ask the physical plan to produce the information they require in their logic and, in turn, physical plans need produce what was requested by instrumentors, and instrumentors should finally consume from it.

Now, compilation under instrumentation involves two challenges not addressed by the traditional producer-consumer compilation model for normal query compilation.

First, recall that normal physical plans are trees. As such, the producer-consumer compilation operates by calling the producer of the root of the tree. In turn, each physical operator node in a physical plan will call the produce functions of its children. Finally, leaf nodes will start producing and will call their parents to consume what they produced for them. In turn, intermediate nodes that consume from their children will also produce for their parents. The process stops when the root operator consumes. Of course, during compilation, data is neither produced nor consumed. Rather, operators generate source code that implements the logic for producing and consuming data. For more details on compilation under normal query execution refer to Chapter 2.

The problem with compilation under instrumentation is that an instrumented physical plan is not a tree but rather a graph where each physical node of the base query has multiple consumers (i.e., consumers of the physical plan and consumers due to instrumentation). To perform compilation of an instrumented physical plan, we have changed the compiler of SMOKE to compile instrumented physical plans with the following strategy:

Before calling the produce function of the root operator, it calls the root of every instrumentor. As soon as an instrumentor calls the produce functions of physical nodes of the base query their compilation stops. When compilation has stopped for all instrumentors, then the compilation of the base query begins. When the physical operator of the base query, from which instrumentors has requested to consume from, can start producing the information requested by instrumentors, then these physical nodes will call the consume functions of the instrumentors (i.e., the ones that we showed in Section 6.3).

Finally, note that in practice the above strategy operates on pipelines of the physical plans. We do so because instrumentors may want information from the execution of previous pipelines in their initialization and overall logic and because there is no point in having an instrumentor initialized at the beginning if it is only going to be used in later pipelines.

### 6.9.2 Physical Algebra

Throughout our discussion in this chapter, we also discussed changes that we made to the underlying physical algebra of SMOKE. More specifically, each instrumentation point, that we introduced in Section 6.3, corresponds directly to a fragment in the implemented logic of each operator. As such, we changed the physical operators to introduce these points in physical operators as points from where instrumentors can consume data flows and integrate their logic. Examples of such minimal changes on the implementation of physical operators are included in Sections 6.3 and 6.4. Finally, instrumentors need to introduce their logic within operators by first registering to instrumentation points. Hence, we extended physical operators with registration functions, as we discussed in Section 6.4.5. We believe these changes are minimal in comparison to rewriting physical algebras every time we need to introduce a technique within a database.

### 6.9.3 Optimizer

Finally, we discuss on a change that we made to the optimizer to account for the declarative (in SQL terms) specification of the instrumentation logic, as we discussed in Section 6.4.3. When the instrumentation logic is expressed as SQL queries, we need to process it as such by the database. These SQL queries, however, involve “sources” that correspond to instrumentation points rather than actual tables. To account for such queries, the optimizer needs to know that an instrumentation point is a source, has a schema, and associated statistics with it. To do so, SMOKE introduces special entries for instrumentation points in its catalog that further associates with statistics (e.g., estimated cardinalities of data flows).

## 6.10 Discussion

Having discussed in detail the components of the Physical Plan Instrumentation Framework, in this section we discuss potential concerns around physical plan instrumentation. More specifically, we first elaborate on potential issues and advantages of alternative database designs (i.e., interpretation and compilation engines) on the implementation of our instrumentation mechanisms on such engines. Then, we discuss security concerns associated with physical plan instrumentation. Finally, we discuss the target audience for our Physical Plan Instrumentation Framework and expected user experiences.

### Interpretation-based query engines

SMOKE is a query-compiled database engine and uses the benefits of compilation for the introduction of instrumentation logic within a physical plan. In contrast, interpretation based engines that do not compile a physical plan, but rather interpret the physical plan, may be limited in their ability to support Instrumentors. To be more precise, for an engine to allow instrumentation of a physical plan, it needs to be capable to change the underlying physical operators at runtime. Based on this observation, we can divide interpretation-based engines into how their underlying runtime allows code modification.

Engines implemented in languages that provide instrumentation features are more naturally amenable to physical plan instrumentation. For instance, physical operators supported by engines that are implemented in Python or Javascript can be easily instrumented.

This is because such languages provide instrumentation mechanisms (e.g., monkey-patching) that allow us to modify code (and by extension physical operators) at runtime.

Engines that have physical operators pre-compiled to machine code (e.g., implemented in C or C++) are harder to instrument. This is because the ways to introduce instrumentation of physical plans in these engines involve low-level operations (e.g., binary instrumentation and changing virtual table entries) that besides being low-level are also specific to the compiler of the language.

Finally, note that a recent trend is to have interpretation engines that deploy just-in-time (JIT) compilation for parts of physical operators. For instance, PostgreSQL 11 uses JIT compilation for expression evaluation and tuple deforming. In this direction, we believe that JIT compilation of the instrumentation logic could follow the mechanisms that we proposed in this chapter for the introduction of compiled and interpreted Instrumentors all while avoiding the difficulties of instrumenting physical operators of pre-compiled engines.

Note that in our discussion above, our focus is on pointing out advantages and disadvantages when considering injection of instrumentation logic within physical operators of interpretation-based engines. This discussion is related to Points, Instrumentors, and the mechanisms we proposed behind them. Other instrumentation operations that we proposed in this chapter (i.e., Actions, Scheduler, Storage Manager, and Announcer) have similar advantages and disadvantages. A main exception regards the Actions component. Actions that modify physical plans at runtime are more efficient and easier to design in interpretation-based engines. This is because in compilation-based engines every modification of a plan needs to be followed by recompilation. This recompilation may incur a significant compilation cost. To mitigate this cost, SMOKE performs ahead-of-time (AOT) compilation of the modified plan (i.e., the plan after Actions will take place). AOT compilation is hard to design, however, because we need to make a decision on when to mitigate the cost (e.g., overlapped with the initial plan execution or along with the compilation of the initial plan that SMOKE currently deploys). In contrast, modifying a physical plan in an interpretation-based engine does not incur such a cost because a modified plan will be interpreted.

### Compilation-based engines

In contrast to interpretation engines, compilation engines can in principle follow the techniques that we discussed in this chapter since SMOKE is a compilation engine itself. However, compilation engines may differ in their design and each such design may have an effect on the actual implementation of an instrumentation framework. Here we highlight how some of such designs can affect such an implementation.

First, multiple compilation engine leverage LLVM [LA04] and LMS [RO10] for compilation of physical plans. This design entails that compilation-based instrumentors (introduced in Section 6.4.2.2) should also be implemented using low-level LLVM or LMS IRs. While there are multiple benefits provided by such IRs and their corresponding compilation frameworks, one potential shortcoming is the lack of easy to use debugging tools. In contrast, SMOKE's internal IR essentially mirrors and gets compiled directly to C++ to provide readability and debugging capabilities (e.g., through gdb or valgrind) of the compiled instrumented physical plans. This is an important difference because, in contrast to implementing physical operators that have fixed and well-defined logic, instrumentation logic can be arbitrarily complicated—rendering readability and debugging mechanisms essential for the development of instrumentation applications.

Furthermore, compilation-based engines may not follow the producer-consumer compilation model in which case compilation of the instrumentation logic needs to be revisited. For instance, recently Tahboub et al. [TER18] proposed a query compilation model under which each physical operator takes as input callbacks. Then, each operator is responsible to produce its results and apply the callbacks on the produced result (which in turn produce their own results). From an instrumentation perspective, this means that instrumentors will need to push their logic within callbacks and the underlying compiler needs to account for the fact that callbacks do not just generate streams of tuples but also the result of the instrumentation. As such, we believe there is ample of space for future work to better understand the design principles of compilation engines that do not only account for compiling queries but also their instrumentation.

## Security

Physical plan instrumentation allows external applications to modify physical plans. As such, it can raise several security concerns. Consider the following examples:

First, consider an instrumentor that wants to read an attribute from a database table, and this instrumentor is run by a user that does not have read privilege on this attribute. As another example, consider the case where an instrumentor has bugs and crushes. Finally, consider an adversarial user that creates an instrumentor to instrument and change the logic of queries posed by other users in adversarial ways.

These examples are just three out of potentially many security concerns that can arise in the context of physical plan instrumentation. Currently, SMOKE does not provide any mechanisms to ensure the resolution of such security concerns. However, SMOKE and its instrumentation mechanisms have been built in such a way to account for future introduction of security. Next, we discuss points in our design specifically targeting the security concerns of the above examples to better propose future work on instrumentation security.

Considering our first example, the underlying storage of SMOKE is only accessible through our Storage Manager. Hence, when a user runs an instrumentor that asks to read portions of the database that the user has no read privilege on, SMOKE can detect this problem through the Storage Manager and internal catalogs holding user privileges. Considering our second example, currently SMOKE executes a compiled instrumented plan as a separate process of the database server process. Hence, if the instrumented physical plan crashes the server remains functioning. This approach is often limited because if the instrumentation logic crashes that should not necessarily mean that the query should also stop its execution (e.g., if provenance capture on a group-by aggregation crushes there is no reason why the group-by aggregation should stop executing). In this direction, we believe future work on continuing executing the query when the instrumentation logic crashes (e.g., through recovering the execution of the query and discarding the instrumentation logic) is also important. Finally, considering our third example, SMOKE supports catalogs hosting user privileges. Extending such catalogs to maintain security rules on instrumentors (e.g., a user cannot instrument the queries of a different user or group) is important future work.

### Target audience and user experiences

The primary target audience of the Physical Plan Instrumentation Framework of SMOKE is database engineers that want to modify physical plans without having to rewrite the database internals. As such, the Physical Plan Instrumentation Framework is by design low-level to account for flexibility and, as such, may be considered as having a steep learning curve. For such an audience, however, a steep learning curve is sensible considering that the alternative is to rewrite the database internals or to find workarounds, as we discussed in Chapter 1. Furthermore, end users of databases are highly unlikely to use such instrumentation frameworks directly. Rather, we believe end users want to use the applications that can be built on top of such frameworks. For instance, it is up to a database engineer to inject provenance capture and analysis within a database (and this is possible through our Physical Plan Instrumentation Framework). End users can then use logical provenance query constructs without worrying on how the capture logic is implemented within the database. In this direction, we believe that understanding the connections between instrumentation applications (e.g., monitoring can be used by online physical database designers, provenance can be used by interactive visualization applications, and logging can be used for lifecycle management and data debugging) and defining the target audience and associated user experiences around each instrumentation application is an important future work to better define the scope and mechanisms that instrumentation-enabled engines need to support.

## 6.11 Conclusions

In this chapter, we introduced a physical plan instrumentation framework that exposes mechanisms that applications can use to implement their instrumentation logic. Throughout our discussion, we outlined several simple techniques across domains (e.g., monitoring, online optimizers, negative provenance managers, and online query optimizers, and physical database designers) and how they can use the provided mechanisms. Connections with other techniques across such domains will be outlined in Part II which evaluates further our mechanisms. In connection with our previous chapters, we noted how provenance capture techniques can push their logic within physical operators through Instrumentors, how to use the Scheduler for injecting and deferring the provenance capture logic, and how to use the Storage Manager for materializing provenance information. Also, combining the mechanisms of the Scheduler with the Announcer provides functionality that future optimizers for provenance capture purposes can consider including online compression of provenance indexes and deciding between INJECT and DEFER semantics at runtime.

# Part II

# Applications

# Chapter 7

# Expressing Interactive Visualizations

In Part I, we described the mechanisms that SMOKE exposes for instrumentation purposes, and we discussed how to express and optimize a provenance manager based on them. The end result is an instrumentation-(and by extension provenance-)enabled database. In this part, we aim to delve deeper into instrumentation-driven application domains to show how to express and optimize their core logic using the mechanisms of our database engine.

In this chapter, we start our discussion over application domains by expressing interactive visualizations within the context of our instrumentation-enabled database engine. More specifically, we first introduce iSQL which is our relational query and data models for the declarative specification of interactive visualizations. Then, we show how iSQL can express several classes of interactive visualizations in a purely relational manner. While iSQL is expressive enough to support the specification of well-known interaction classes, we show why a purely relational approach leads to specifications that are hard to express and optimize. To address this problem, we show how to express data-intensive interaction classes (i.e., multi-view linking, interactive selections, and logic over selections) using our provenance- and instrumentation-related capabilities. As a result, we can cast their optimization into optimizing provenance and instrumentation constructs in our database engine.

## 7.1 Introduction

Interactive data visualizations enable users to rapidly recognize important patterns within the data, by leveraging the powerful capabilities of the human perceptual system, and to identify and explore salient relationships that are not readily evident from a static visualization. As such, they constitute a cornerstone in many human-in-the-loop data analysis and management systems across domains including data exploration and decision-support [Ora14; Pow18], knowledge exploration [TSW11; ADM+15], debugging and analysis of machine learning and statistical models [Ten16; RSt16; SGB+18], interactive data cleaning [KPHH11; KPP+12; WM13; WMS12] and profiling [EEI+13; PBF+15], to name a few.

The increasing importance and ubiquity of interactive visualization tools, along with the massively increasing scale of modern datasets, has seen a convergence between the visualization and database communities. Visualization systems [SRHH15; SMWH17; BOH11] incorporate data processing capabilities such as filtering, grouping, aggregation, ordering, and scaling in order to compute data summaries that are further rendered on the screen. However, increasing dataset sizes has caused data processing to become a core bottleneck that impedes interaction responsiveness. To illustrate, consider the multi-view interactive visualization in Figure 7.1:

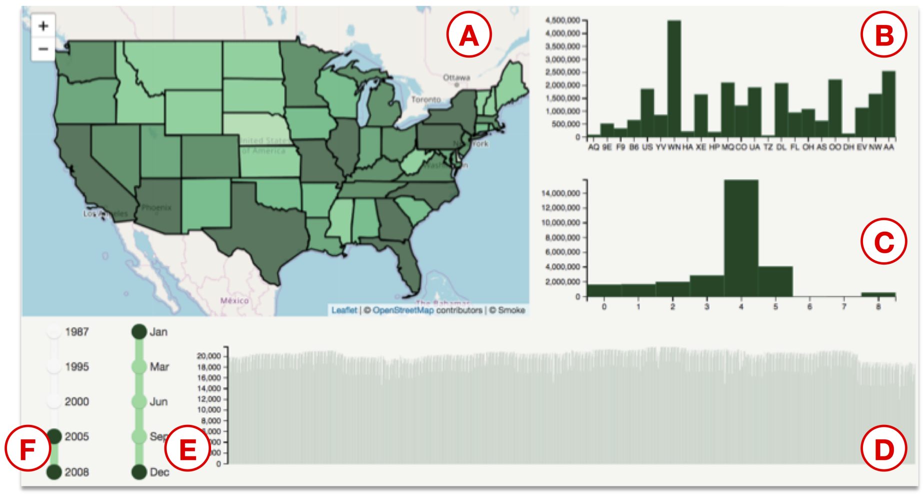

A
B
C
D
E
F
1987
1995
2000
2005
2008
Jan
Mar
Jun
Sep
Dec
AQ 9E F9 B6 US YV WN HA XE HP MQ CO UA TZ DL FL OH AS OO DH EV NW AA
Leaflet | © OpenStreetMap contributors | © Smoke

Figure 7.1: Example of an interactive visualization.

**Example 11 (Exploring Flight Delays)** Figure 7.1 visualizes a breakdown of delayed flights [Ont] coupled with a crossfilter interaction technique [cro15]. Each chart renders the output of a count aggregation of delayed flights grouped by different attributes: by state **A**, airline **B**, departure delay **C**, date **D**, month **E**, and year **F**. Thus, the visualization may be modeled as a large relational workflow composed of these aggregations, along with visualization workflows to map the results to visual marks (e.g., rectangles and polygons) which, in turn, are mapped to pixels on the screen. Crossfilter interactions let users select data in any of the views and see the other views update to show the statistics represented by the selected subsets. For instance, an interactive range selection on the years (F) triggers the re-execution of the aggregations, this time considering only records in the selected range [2005, 2008], and the update of charts to reflect only the aggregations in this time range.

The main characteristic of interactions is that humans are on the critical path of data analysis and non-interactive response latency (e.g., >150ms) to their interactions has a detrimental effect on their overall data analysis [Shn84; Han12; HS12]. For instance, consider that the response to the interactive selection of years [2005-2008] in our example takes more than a second to update the charts. Also, assume that throughout this time users observe the interface that still reflects the initial group-by aggregations (i.e., the visual pane has not been updated to reflect there is a process going on). As soon as the response time exceeds interactive latencies (e.g., 150ms), users will start inferring that either all flights lie between the years [2005-2008], or more generally that flights from other years have no impact on the overall trends, or that the application has crashed. All three scenarios have a negative effect on the overall analysis and highlight the importance of interactivity.

Drawing the connection between relational workflow processing and interactive visualization not only improves the productivity of developers by introducing higher level languages to express visualizations, but has led to a rich area of performance-oriented database research with the goal to increase the interactivity and user engagement with front-ends. For instance, recent research efforts adapt query optimization techniques to the visualization domain and develop novel techniques inspired by unique characteristics of visualizations. These include adapting columnar execution [KPP+12], perception- and visualization-aware online aggregation [PSWC17; AW16; KBP+14; RAK+17], speculative exploration sampling [KJTN14], and visualization prefetching [BCS16], to name a few. Notably, most of this work has been focused on speeding up specific visualization interactions or specific classes of database queries.

In this chapter, we build on this convergence in two-folds: First, we introduce iSQL which is our relational approach towards the specification of interactive visualizations. By expressing interactive visualizations in relational terms (i.e., with SQL-like constructs and relational data models) we can bring the optimization technology of databases to the interactive visualization domain, which is dominated by manual implementations with ad-hoc and error-prone optimizations. Having described iSQL, then we highlight the connection between provenance and instrumentation with visualization interactions. More specifically, we show that specifying interaction classes in purely relational terms leads to specifications that are hard to express and optimize. To address this problem, we extend iSQL with provenance constructs and accounting for the provenance- and instrumentation-aware capabilities of SMOKE to show that common interactions that are hard to express and optimize in purely relational terms, can be naturally expressed and optimized in a blend of relational, provenance, and instrumentation terms within our database engine.

## Contributions and Chapter Outline

In the rest of this chapter, we start with the necessary setup (Section 7.2). Then, we present our contributions as follows:

- We introduce iSQL, a relational data model and language for expressing interactive data visualizations (Section 7.3).
- We evaluate the expressiveness of iSQL along several classes of interactions from well-known interaction taxonomies. (Section 7.4)

- We explain why classes of data-intensive interactive visualizations are hard to express and optimize in relational terms. To address this problem, we draw the connections between interactive data visualizations with provenance and instrumentation per class. Based on these connections, we show how we can express these classes in a blend of relational, provenance, and instrumentation terms. (Section 7.5)

## 7.2 Setup

To ease our discussion throughout this chapter, we use the following database of delayed flights to build static and interactive data visualizations and explain our overall techniques.

```
flights(fid,y,m,d,h,adelay,ddelay,origin,dest,carrier)
airlines(carrier,name,iata,active)
airports(apid,name,iata,lat,lon,elevation,city,state)
states(state,name,polygons[])
```

Figure 7.2: The flights database schema.

The `flights` table records delayed flights: for each flight with id fid, `flights` records its arrival and departure delays (i.e., `adelay` and `ddelay`, respectively) from a source airport with id `origin` to a destination airport with id `dest` along with the departure time of the flight (i.e., **y**`ear`, **m**`onth`, **d**`ay`, and **h**`our`) and the airline that operated the flight `carrier`. The `airports` table records the id of each airport (`apid`) along with its `name`, `iata`, latitude (`lat`), longitude (`lon`), `elevation`, `city`, and `state`. The `airlines` table stores the id of an airline (`carrier`) along with its `name`, `iata`, and whether or not the airline is still `active`. Finally, the `states` table records the `state` code, the name of the state, and a multidimensional array of `polygons` that corresponds to the geographical bounds of states in the US. Colored attributes correspond to pk-fk relations.

## 7.3 iSQL: Data Model and Language Overview

In this section, we present iSQL, our relational approach towards the specification of interactive data visualizations. More specifically, we first express static visualizations in a relational manner (Section 7.3.2). Then, we introduce user interactions as event streams (Section 7.3.3). Finally, we use these concepts to express interactive data visualizations as database queries involving joins between event streams and static visualizations (Section 7.3.4). To ease our discussion in this section, we start by presenting a linked-brushing example. (Section 7.3.1).

### 7.3.1 Linked-Brushing Example

To better illustrate key points of our data model and language we use a simple linked brushing [BC87] example that most visualization toolkits and systems implement imperatively.

```
Delays = SELECT     AVG(adelay) AS avg_adelay,
                    AVG(ddelay) AS avg_ddelay,
                    AL.carrier AS carrier
         FROM       flights AS F, airlines AS AL
         WHERE      F.carrier = AL.carrier
         GROUP BY   AL.carrier
```

Figure 7.3: Average arrival (`avg_adelay`) and departure (`avg_ddelay`) delays per carrier materialized in the relation `Delays`.

**Example 12 (Linked Brushing)** Consider the SQL query in Figure 7.3 and the linked brushing example in Figure 7.4. The query in Figure 7.3 materializes the the average arrival (`avg_adelay`) and departure (`avg_ddelay`) delays per carrier. Figure 7.4 visualizes the `Delays` data and shows step-by-step an application of linked brushing: initially, a static visualization is composed of a scatterplot that correlates the average arrival and departure delay of each carrier, and a histogram that shows the number of flights for each carrier. (Step 1) shows a mouse drag interaction that selects a rectangular region in the scatterplot, alongside all marks (i.e., circles) inside the selection. The subset of carriers that correspond

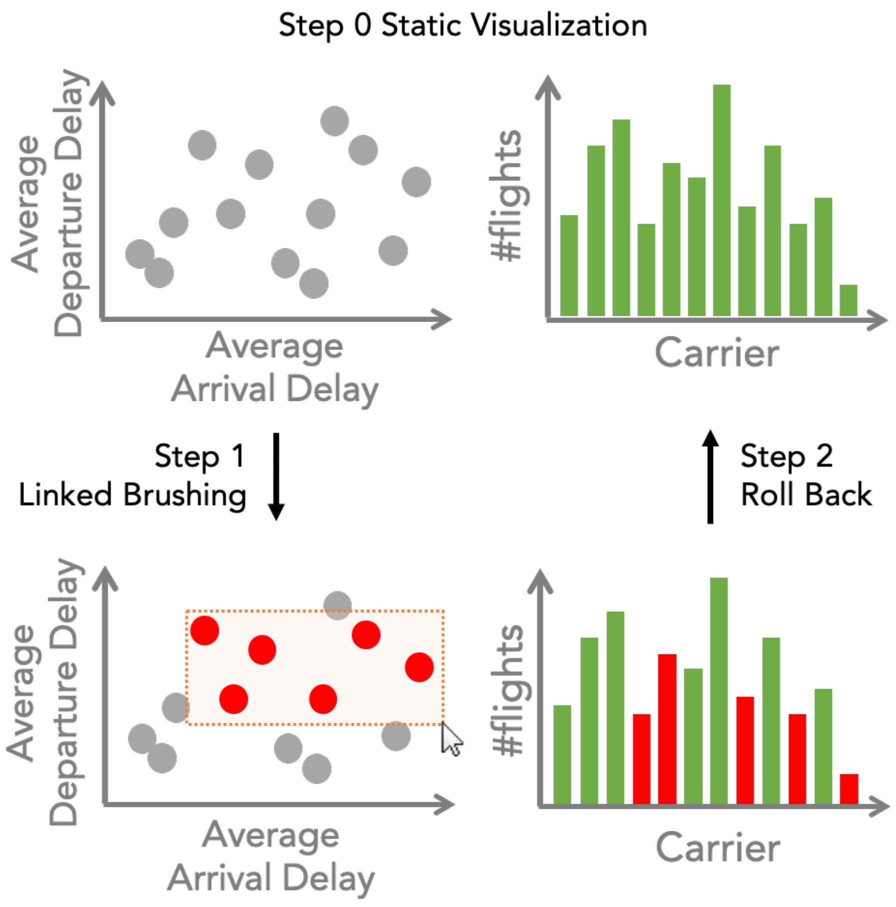


Figure 7.4: Brushing and linking example using the `Delays` relation.

to the highlighted circles is "selected" by turning red, as are the histogram bars corresponding to selected states. Post-selection, the user may reset the visualization (Step 2), keep the points highlighted, or perform another selection.

The goal of this section is to show how such data interactive visualizations can be composed in a purely relational manner to avoid imperative implementations and to optimize them from within a database. Next, we introduce the data model and language overview of iSQL for the specification of static data visualizations, that follows prior work, and how we extended them for the support of interactive data visualizations.

### 7.3.2 Static Visualizations

In line with prior work [Wic09; SMWH17; WBM14], we model a static visualization as a mapping from a set of database relations $\mathcal{R} = \{\mathrm{R}_1, \ldots, \mathrm{R}_{|\mathcal{R}|}\}$ in the data domain to relations in the visual domain. Furthermore, we model the visual domain using two types of relations: *Marks* and *Pixels* relations. Next, we give a brief description of the two relation types as well as the notion of the mappings in iSQL.

**Marks.** The former type of relations (i.e., Marks) describe shapes (e.g., lines, circles, and polygons) to be rendered in the visualization. Each such relation type corresponds to a specific mark type (e.g., line, circle, or rectangle), with attributes including the geometry and visual encoding of the corresponding marks. Furthermore, visual representations that have a graph-like structure (e.g., dendrograms or trees) can be encoded as facts over Marks relations that constitute dimensions. For instance, a tree can be modeled by a Marks relation involving circles (i.e., to encode the nodes of the tree) whereas the connections between nodes can be materialized in a separate (fact) relation. Finally, visualizations that are composed out of multiple visualizations (e.g., widgets or even complicated dashboards) follow a similar data model with primitive marks encoded in (base) Marks types of relations and connections between them are encoded in fact relations.

**Pixels.** The latter type of relations (i.e., Pixels) models the rasterized pixels shown to the user. More specifically, Pixels is a special relation `P(x,y,R,G,B,A)` that models the color (`R,G,B`) and transparency (`A`) encodings at every pixel coordinate (`x,y`) of a given screen.

**Mappings.** Finally, the mapping of the static visualization encapsulates the data transformations (e.g., aggregation and scaling) that encode data summaries as geometry and visual encodings (e.g., height and color of a circle). Since, both the data and visual elements are represented under the relational model, the mapping is expressed using relational queries. However, note that while Marks and Pixels are declared in the relational model, their contents may not be persisted in the database, if not needed. Instead, they can be projected and maintained by the underlying rendering devices.

```
SPLOT_POINTS =
    SELECT 8 AS radius,
           'gray' AS stroke,
           'gray' AS fill,
           linear_scale(Delays.avg_adelay, scale_x) AS center_x,
           linear_scale(Delays.avg_ddelay, scale_y) AS center_y,
           carrier
    FROM  Delays, scale_x, scale_y;
scale_x = SELECT MIN(avg_adelay), MAX(avg_adelay) FROM Delays;
scale_y = SELECT MIN(avg_ddelay), MAX(avg_ddelay) FROM Delays;
P = render(SELECT * FROM SPLOT_POINTS);
```

Figure 7.5: Static visualization for the scatterplot of our example.

,

To illustrate our formalism of static visualizations, consider the iSQL code snippet in Figure 7.5. It shows a query that maps the flight delays data from the relation `Delays(carrier, avg_adelay, avg_ddelay)` of Figure 7.3 to a Marks relation `SPLOT_POINTS` that represents the circles of the scatterplot in Figure 7.4. Each projection clause defines an attribute of the circle mark, such as the `radius` as well as `stroke` and `fill` colors. The `linear_scale` UDFs linearly transform the departure and arrival delays to their corresponding pixel coordinates—the UDF uses the `scale_x` and `scale_y` relations, which include the minimum and maximum values of the `avg_adelay` and `avg_ddelay` attributes, to compute the transformation. The last attribute, namely, `carrier`, is typically used in visualizations as a way to ensure a correspondence between the rendered mark and the input record. This is required to support responses to user interactions, as we will see in Section 7.3.3. (In Section 7.5, we explain why these annotations introduce problems and present provenance extensions that enable this correspondence in a declarative manner.) Furthermore, Marks relations are rendered using the `render` table UDF. (Note that iSQL supports both table and record UDFs. However, they are restricted to pure functions without side effects except rendering functions that may produce visual side effects.) Finally, note that similar queries and rendering functions can be used to define the static visualizations of the histogram and axes in Figure 7.4.

The above follows the procedures described in [WBM14] and illustrates how static visualizations can be expressed as database views. Next, we describe our extensions towards the specification of interactive data visualizations.

### 7.3.3 User Interactions

Interactive visualizations enable exploration capabilities by executing queries in response to user interactions. To encode this process, iSQL models user interactions as streams of low-level events (e.g., mouse down, move, and up) out of which we can extract compound events (e.g., mouse drag) as patterns over the streams of low-level events. Besides the decoupling of the logical event representation from its physical (e.g., in browsers and GUI frameworks), this representation has two important properties. First, it allows us to model responses to user interactions as queries involving event streams, visual elements (encoded as relations in Section 7.3.2), and base relations. Second, it allows us to draw a direct analogy between an interaction and a database transaction: each compound event, that defines a complex interaction, may either transition the database to a new version or rollback to the version right before the beginning of an interaction. This functionality is important for the specification of complicated interaction techniques (e.g., undo and redo).

To capture event streams of low-level events, we adopt the data model of CQL [ABW03]: given an alphabet of low-level events $\Sigma$ (e.g., $\Sigma = \{\text{mouse\_down, key\_press}, \ldots\}$), we can model a stream of them as an (unbounded) set of ordered pairs $\langle \mathrm{s}, \mathrm{t} \rangle$, where $\mathrm{s} \in \Sigma$ and $\mathrm{t}$ is the time when a user performed $\mathrm{s}$. Each symbol in the alphabet (e.g., mouse down) is defined as a relation with a meaningful schema (e.g., mouse_down can have attributes $\mathrm{x}$ and $\mathrm{y}$ to encode where the mouse_down event took place). The schema of the stream is the set union of the schemas of individual events.

To extract compound events from low-level event streams, we could leverage a number of automata-based approaches that identify complex patterns in event streams [CCD+03;

```
C = EVENT MOUSE_DOWN AS D, MOUSE_MOVE* AS M*, MOUSE_UP AS U
    WHERE FORALL m IN M m.y > 5
    RETURN (D.t, D.x, D.y, 0 AS dx, 0 AS dy),
           (M.t, D.x, D.y, (M.x - D.x) AS dx, (M.y - D.y) AS dy)
```

Figure 7.6: Event statement to generate a compound event stream.

ABW03; WDR06; SRHH15; KHDA12; JMS+08]. For instance, regular expression-based languages such as Proton [KHDA12] are used in the user interface literature to recognize user gestures. We borrow these ideas in a sequence matching language similar to SASE [WDR06], which compiles patterns into non-deterministic finite automata (NFA).

The iSQL code snippet in Figure 7.6 shows the event statement that specifies the user drag interaction of our example in Figure 7.4. It does so by defining a compound event stream `C` as a sequence of mouse down, repeated mouse move, and mouse up events. This sequence is compiled into an NFA. Non-matching event types (e.g., a key press), as well as events that fail predicates in the `WHERE` clause, are filtered from the input stream and not processed by the NFA (e.g., a `D.y > 1` predicate could remove mouse down events below `1` pixels from the input stream). Existential and universal quantifiers transition the underlying NFA to a reject state upon failure (e.g., a mouse move event with M.y=4 would transition the NFA to a reject state due to the universal quantifier `FORALL` in Figure 7.6). Finally, the `RETURN` clause defines a sequence of union-compatible projection statements, and concatenates all statements that can be evaluated by the matching events. For instance, the iSQL code snippet in Figure 7.6 first emits the mouse down event, followed by each move event along with its distance from the down event.

As a concrete example, Table 7.1 illustrates the state of the relation `C` during a user's drag movement. The mouse down at `t=0` inserts the first record, based on the first projection statement of the `RETURN` clause. Note, that no record is inserted in `C` at `t=0` due the second projection statement of the `RETURN` clause. This involves mouse move events that have not happened yet. Subsequent mouse move events insert corresponding records into `C` based on the second projection statement of the `RETURN` clause. For every mouse move, no record is inserted in C due to the first projection statement because it does not involve mouse move events. Finally, the mouse up event transitions the NFA to an accept state and terminates insertions into the relation C. (Note that no record is inserted in C due to the mouse up event because it is not involved in any projection statement.)

Under this model, an EVENT statement defines the boundaries of an atomic user interaction in a direct analogy to "transaction" boundaries. A start state of the EVENT NFA begins the transaction while accept states commit possible changes to the database triggered by insertions into event tables. To prevent never-ending transactions, we constrain EVENT statements to sequences that end with a non-repeating event, and the underlying NFA can transition only once to an accept state and commit a transaction. Reject states of the NFA lead to aborting the transaction. By default, whether we commit or abort, the event stream will be cleared to initiate new user interactions. Finally, note that the main difference from traditional transactions is that the "uncommitted" state, such as the state of Marks relations throughout the mouse move events in our example, can be exposed to the user in the form of visualization updates, as we will see with interactive visualizations next.

### 7.3.4 Interactive Visualizations

Interactive visualizations can respond to user interactions by expressing their logic as queries involving event streams, base data, and visual elements (encoded as relations

| t | x | y | dx | dy | Input event |
|---|---|---|---|---|---|
| 0 | 5 | 15 | 0 | 0 | MOUSE_DOWN(0,5,15) |
| 1 | 5 | 15 | 1 | 2 | MOUSE_MOVE(1,6,17) |
| ... more MOUSE_MOVE events ... | | | | | |
| 40 | 5 | 15 | 5 | -5 | MOUSE_MOVE(40,10,10) |
| MOUSE_UP(41,10,10) terminates the query | | | | | |

Table 7.1: Contents of the event table C in our example after a potential sequence of low-level events.

```
selected     = SELECT SP.carrier
               FROM   C, SPLOT_POINTS@vnow-1 AS SP
               WHERE  in_rectangle(bbox(C), SP);
SPLOT_POINTS = SELECT ..., 'gray' AS fill
               FROM Delays, scale_x, scale_y
               WHERE carrier NOT IN selected
               UNION
               SELECT ..., 'red' AS fill
               FROM Delays, scale_x, scale_y
               WHERE carrier IN selected;
```

Figure 7.7: Selection interaction for the example scatterplot.

in Section 7.3.2). To illustrate, the iSQL code snippet in Figure 7.7 shows how iSQL can declaratively specify our linked brushing example.

To start off, we specify the set of `selected` marks using a join between `C` and the scatterplot Marks relation `SPLOT_POINTS`, as shown Figure 7.7: `bbox` is shorthand for a query that computes the selection box of the mouse drag events in `C`, and `in_rectangle` is shorthand for a predicate that checks whether a mark `SP` intersects with the selection box. As the event query populates `C`, the `selected` relation updates accordingly. By defining the Marks relation for the scatterplot to perform different projections for selected and non-selected records, we can express brushing. Similarly, we can define a Marks relation for the histogram of our example. This is possible because both Marks relations coordinate on the `selected` relation we have the desired effect of linked brushing.

The main challenge in expressing interactive visualizations is that the underlying workflows typically contains cycles. The `selected` view, for instance, depends on the `SPLOT_POINTS` view and vice versa. To address this issue, iSQL allows referencing past versions of relations in queries. Specifically, developers can specify the committed state of a relation i interactions (or transactions) ago by adding the suffix `@{vnow-i}` to a relation name. The syntax `@{tnow-j}` specifies the state of a relation j events ago within the current interaction (or transaction). For example, Figure 7.7 computes the selected marks by performing hit testing on the Marks `SPLOT_POINTS@{vnow-1}`, which is the version of `SPLOT_POINTS` at the beginning of the current interaction as expressed by the event statement in Figure 7.6. This approach highlights the relationships between transactions and interactions and simplifies support for popular techniques such as undo or mouse trails.

## 7.4 Expressiveness of iSQL

So far we have introduced the fundamental concepts of iSQL, and now we seek to evaluate its expressiveness. iSQL is an extension to SQL, and any data processing required during initial workflows, workflows initiated due to user interactions, or visualization workflows can make use of the full expressiveness of SQL. This leaves us with the question of what interactive visualization techniques are expressible in iSQL. Similarly to previous interaction grammars [SMWH17], we show that iSQL can express interactions as classified in current interaction taxonomies. More specifically, next we discuss how iSQL expresses interaction techniques under each category of the Yi et al. [YKSJ07] taxonomy (i.e., select, explore, encode, abstract/elaborate, filter, reconfigure, and connect).

### Select

The first category, namely, *select*, includes interaction techniques that can mark something as interesting. As we showed in our linked brushing example, iSQL introduces interactive selections and direct manipulation of circles (e.g., change the color of selected marks) to differentiate the selected from the non-selected circles. Such interactive selections constitute fundamental building blocks of interactive applications because they initiate and drive the post-interaction exploration logic [SMWH17; NS00; Wil03].

### Explore

The second category, namely, *explore*, indicates the ability to explore different subsets of data. In this direction, recall the semantics of the `EVENT` statement and the updates of the visualization states. When a user interacts with a mouse down, mouse move, mouse up sequence of events, a subset of the data is selected. When the user starts issuing another sequence of mouse down, mouse move, mouse up the selection changes and the user can explore another subset of the data. As another important example in the explore category, consider a panning interaction for our example (i.e., moving the camera across a scene or change the scene while keeping the camera still). Previously, we updated the `fill` attribute of the circles to visually indicate the selection. Instead, consider an update of the `center_x` and `center_y` attributes based on the mouse move events of the `EVENT` statement. Since the scatterplot lies within a fixed viewport of the visual space graphically translating the circles (i.e., updating their positions) would result in moving some circles out of the scene while some others could enter the scene. This results in the intended panning effect which allows the user to explore subsets of data hidden due to the constrained visual space.

## Abstract/Elaborate

The third category, namely, *abstract/elaborate*, includes techniques that adjust the level of abstraction of data representation. An important technique in this category is the generation of tooltips. In our example, when we select a circle in a scatterplot we may need to show a tooltip that contains the actual arrival and departure delays for the carrier. In this direction, recall how we annotated each circle with the `carrier` and used this information to trace back the selected marks and identify this piece of information. Using this information, we can then create a tooltip as a Marks relations. As another example in this category consider semantic zooming. To support semantic zooming we need to change between views during a zoom-in interaction. In our example, consider a zoom in on the bar chart that breaks down the number of flights of the carrier to the number of flights of the carrier grouped by state by changing the histogram to a stacked histogram (i.e., each stack reveals the number of flights per state). One way to accomplish this functionality is to union the histogram (initial view) with the stacked histogram (zoomed-in) and change between them based on user interactions. (Note that both the histogram and the stacked histogram are Marks relations specifying rectangles. Hence, they can be unioned).

### Filter

The fourth category, namely, *filter*, includes interaction techniques that change the data being presented based on some condition. Specifying these conditions (e.g., range and equality) in SQL is straightforward as parameterized selection predicates. Visually, users instantiate conditions using dynamic query controls or widgets. Widgets are graphical elements such as sliders, radio buttons, drop down selections lists, or textboxes that users can interact with (e.g., adjust the pins on a slider) to specify a condition. For instance, consider the barchart in our example that shows the number of flights of each carrier and a contiguous slider where users can select a range of number of flights by moving the slider pins. The intended effect is to hide the bars that correspond to carriers with number of flights not in the selected range. As sliders are graphical elements, they can be specified using iSQL as database queries. Interactions with the pins of the sliders result in the change of the positions of the pins. The problem is that for every position of the pins we want to invert them so that from the position of the pin to identify the underlying number of flights. These inversions, typically called scale inversions [SRHH15], are supported in iSQL using inverse mappings [WS97]. Based on the results of the inversion (i.e., a range of number of flights) we can update the Marks for the histogram by projecting the bars for carriers with number of flights above the range, similarly to what we did with color changes for our linked brushing example.

### Connect

The fifth category, namely, *connect*, consists of interaction techniques that are used to (1) highlight associations and relationships between data items and (2) show hidden data items that are relevant to a specified item. In Section 7.3, we showed how to accomplish a brushing and linking effect that associates the selected data items in the scatterplot with their corresponding data items in the histogram. Furthermore, we also discussed how using lineage operators we can identify hidden data items related to a specified item (e.g., for tooltips generation).

## Reconfigure

The sixth category, namely, *reconfigure*, includes interaction techniques that change the arrangement of a data visualization (e.g., sorting or rearranging columns in a table view, the baseline adjustment feature in a stacked histogram, or normalization of data points in a line chart). Here, we discuss the normalization of data points as an interesting example. Consider again the scatterplot `SPLOT_POINTS` of our example and a normalization interaction that changes the positions of selected data points based on a normalization function. The normalization function can be registered in iSQL as a pure UDF. To enable the reconfiguration, we can change the position of the scatterplot circles by applying the normalization on the attributes `avg_adelay` and `avg_ddelay`; the scales `sx` and `sy`; or directly on the circle positions, depending on the normalization semantics.

## Encode

The seventh and final category, namely, *encode*, consists of techniques that can alter the visual representation of the data. A subset of these techniques changes the geometry and the visual encoding of visual elements. In our data model, we have introduced Marks relations whose attributes include the geometry and the visual encoding of the corresponding marks. As we showed in our example, we changed the color of a circle by changing the `fill` attribute of the `SPLOT_POINTS` based on user interactions. Furthermore, to update a color of the scatteplot from a palette of colors, which is an important technique in this category, we can first define the palette as Marks (e.g. a set of rectangles with a different color attribute). Then, we can define a selection on these marks and update the `fill` attribute. Similarly, we can update the size, orientation, or font of Marks.

## 7.5 Connections with Provenance and Instrumentation

Having shown the expressiveness capabilities of iSQL without provenance and instrumentation. Now we turn, to show what classes of interactions can be expressed with iSQL, provenance, and instrumentation. More specifically, we start by extending our initial linked brushing example to account for a more complicated visualization scenario (Section 7.5.1). Then, we introduce the connections between instrumentation and provenance with three classes of interactions: multi-view linking (Section 7.5.2), interactive selections (Section 7.5.3), and logic over selections (Section 7.5.4). In our discussion, we discuss how these classes are related with the classes of the Yi et al. taxonomy, we a) show difficulties on both expressing and optimizing these classes in pure relational terms, b) address them through the connections with instrumentation and provenance, and c) make notes on performance and semantics that we believe developers need to be aware of when implementing visualizations in instrumentation-(and by extension provenance)-enabled database systems.

### 7.5.1 Initial Static Visualization Extended

Let us start by extending our initial static visualization Figure 7.4 to also create a visualization that depicts the number of flights for active airlines per state as a heatmap. In iSQL terms, we can specify this visualization as shown in the iSQL code snippet in Figure 7.8.

`SC` specifies the data processing part of the visualization and consists of a join between the `flights`, `airlines` filtered to only active ones, and `airports` relations followed by a group by state count aggregation. (`SC` also computes the average departure and arrival delays per state that we use later in interactions.) `M` constitutes part of the visualization workflow that transforms the output of `SC` into attributes of polygon marks (i.e., geometry and color of each polygon). `color()` is syntactic sugar for an equation that maps each count value to an output range of green hues, where the input range is computed by `S` as the minimum and maximum counts from `SC`. Finally, the polygons are rendered on the screen using a mark-specific `render_map()` shim, as we discussed in Section 7.3.

```
-- Data Processing
SC = SELECT   COUNT(*) AS cnt,
              AP.state AS state,
              AVG(adelay) as avg_adelay,
              AVG(ddelay) as avg_ddelay
     FROM     flights AS F, airports AS AP,
              airlines AS AL
     WHERE    F.carrier = AL.carrier AND
              F.origin = AP.apid AND
              AL.active = 'Y'
     GROUP BY S.state
-- Visualization
S = SELECT MIN(cnt) AS mi, MAX(cnt) AS mx FROM SC
M = SELECT states.polygons,          -- geometry
           color(SC.cnt,S.mi,S.mx)   -- color
    FROM   SC, S, states
    WHERE  states.state = SC.state
P = render_map(M)
```

Figure 7.8: Example of a static visualization.

Under this model, the overall static visualization (depicted in Figure 7.14) is a complex relational view that maps the input database in *data space* to rendered marks in *pixel space*. More specifically, the first relational workflow maps the input data to a heatmap, the second maps data to a histogram, and the last one maps data to a scatterplot. For convenience, we will also refer to these relational workflows as $V_1$, $V_2$, and $V_3$, respectively. Next, we elaborate on the connections of common interactive capabilities with instrumentation and provenance concepts by building on this static visualization example.

### 7.5.2 Multi-View Linking

Linking is a common class of interactions where selections in one view update other views. Prominent examples, as we have already discussed, include linked brushing and cross-filtering. In terms of the Yi et al. taxonomy, linked brushing corresponds to the Connect

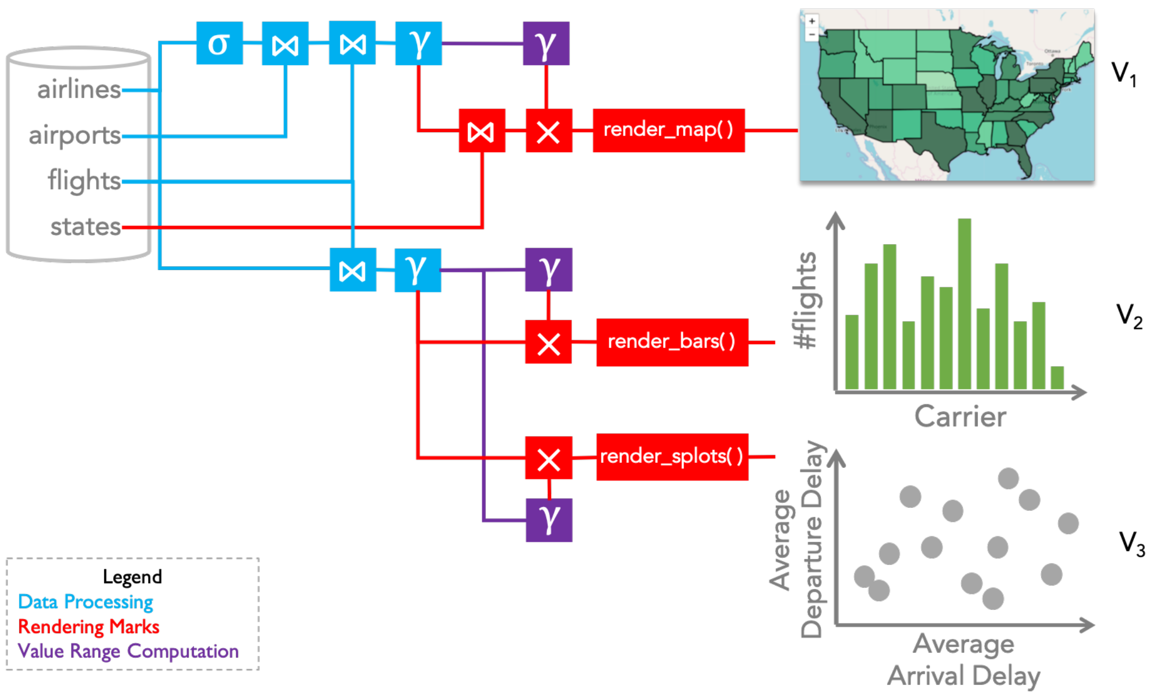


Figure 7.9: Static visualizations broken into data processing, value range computation, and mark rendering operators for our three example visualization views.

class whereas cross-filtering is a technique that spans multiple classes including Connect, Filter, and Abstract/Elaborate. Furthermore, all linking interactions are connected with the Select class since linking is triggered post-selections.

## Linked brushing

Consider again our linked brushing example between the scatterplot and the histogram that we discussed in Figure 7.4. More specifically, recall that we annotated each mark in the scatterplot with each corresponding `carrier` to encode the input-output relationships, so that we can support the linked brushing effect in response to user interactions. Another way could have been to annotate marks with ids and materialize the connections between circles and delays (`circles` $\rightarrow$ `delays`) as well as histogram bars and delays (`delays`

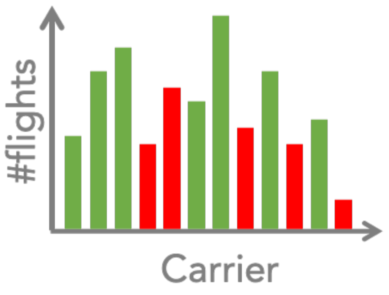
$\rightarrow$ `bars`) in separate tables. Given a selection of circles, we could use the mapping table `circles` $\rightarrow$ `delays` to find the subset of delays corresponding to selected circles and then use the identified delays to go to histogram bars using `delays` $\rightarrow$ `bars`. Both ways, while they can express linked brushing, are problematic.

To explain why, let us first note that linked brushing, as an operation, is expressible in provenance terms. First, we want to go back from the selected circles to the input delays. This is a backward trace query from the circles `SPLOT_POINTS` to the input `Delays`. Then, we want to go forward from the subset of backward traced delays to the bars to highlight them. This is a forward trace query from the subset of delays to the bars. Highlighting of bars by changing their color from green to red can be expressed as a provenance consuming SQL query that updates the color of the forward traced subset of bars.

Now, to see why both approaches above are problematic, recall from our discussion in Chapter 3, the logical provenance capture approaches. The first approach that we discussed above and used in Section 7.3 is the logical denormalized approach for provenance capture that annotates output relations with input ids. The second approach with the mapping tables is exactly the logical normalized approach that stores provenance in provenance relations. As we discussed and showed experimentally in Chapter 3 , both approaches slowdown both the provenance capture and the provenance querying phases. In interactive visualizations this is translated to slowdown of the initial static visualizations (query execution + logical provenance capture) and interaction (provenance querying). Furthermore, recall from our discussion in Chapter 3 that the first approach (i.e., logical denormalized) does not even have a simple way to perform forward tracing besides residing to lazy provenance query evaluation or spending extra time for indexing purposes.

The discussion above leaves us with the question, how can we express linked brushing in a provenance-enabled system such as SMOKE. We start by showing how to express linked brushing using only backward trace, leaving the forward trace and the update to purely SQL terms. Then, we discuss how to express the forward trace and update. We conclude by discussing a general model for expressing linked brushing in complicated views.

```
B =
    BACKWARD TRACE
    FROM SPLOT_POINTS@vnow-1 AS SP, C
    WHERE in_rectangle(bbox(C), SP)
    TO Delays;

SPLOT_POINTS =
    SELECT ..., 'red' AS fill
    FROM B
   UNION
    SELECT ..., 'gray' AS fill
    FROM (Delays MINUS B);

HIST_BARS =
   SELECT ..., 'red' AS fill
   FROM B
  UNION
   SELECT ..., 'green' AS fill
   FROM (Delays MINUS B);
```



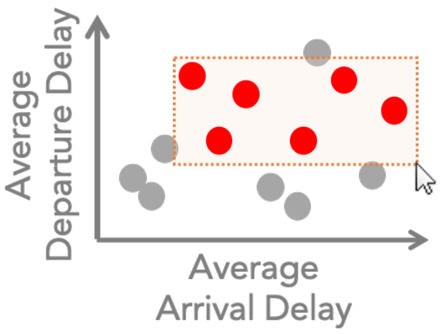


Figure 7.10: Expressing linked brushing declaratively using backward trace.

Consider the iSQL code snippet in Figure 7.10. The `BACKWARD TRACE` query is a provenance statement that traces backward a subset of the output to the subset of its contributing inputs. (This is the same with backward provenance query that we discussed in Chapter 3. Here we give it a structure so that we can express the selection of the output subset.) Its structure resembles the structure of a `SELECT` query. More specifically, the `FROM` clause along with the `WHERE` clause denote a join among the event stream `C` and the scatterplot that determines the selected circles. The `TO` clause denotes the relation to trace backwards from the result of the join (i.e., `Delays`). The interactive histogram and scatterplot are defined based on the partition $\{$`Delays`$\backslash$`B`, `B`$\}$: circles and bars for the backward traced subset `B` are colored red; the unselected marks in `Delays`$\backslash$`B` are colored gray (circles) and green (bars). Hence, events on the stream `C` change the backward traced subset that, in turn, changes the bars and circles with the desired linked brushing effect.

```
B =
    BACKWARD TRACE
    FROM SPLOT_POINTS@vnow-1 AS SP, C
    WHERE in_rectangle(bbox(C), SP)
    TO Delays;

SPLOT_POINTS =
    FORWARD TRACE B TO SPLOT_POINTS@vnow-1
    SET color='red'

HIST_BARS =
   FORWARD TRACE B TO HIST_BARS@vnow-1
   SET color='red'
```

Figure 7.11: Linked brushing using both backward and forward tracing.

In the specification above we expressed linked brushing using only the backward trace provenance statement. The forward trace, however, has been implemented in relational terms by recomputing each output visualization view. We can also express this functionality using a forward provenance statement followed by an update, as shown in Figure 7.11. The `FORWARD TRACE` statement traces forward the backward tracked subset of delays to circles and bars, and updates the forward traced subset by updating their color to red.

Finally, note that our discussion above is limited to going back from selected scatterplot points to the `Delays`. This is a fairly simple scenario with data pre-materialized in the `Delays` relation and the static visualization involved no data processing. As we showed with our example in Figure 7.8, however, static visualizations can be arbitrarily complex. In such cases, implementing linked brushing interactions in SQL terms is even harder to express and optimize because we need to account for all the complexities of the underlying visualization workflow. However, in provenance terms we only need to express what to trace back and what to trace forward. For instance, Figure 7.12 shows brushing and linking by

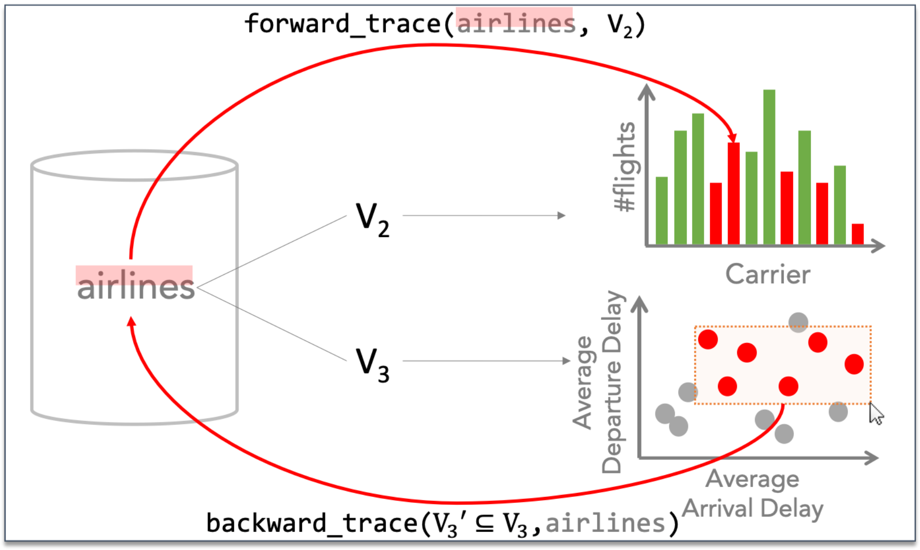


Figure 7.12: Linked brushing using backward and forward trace statements over complicated views.

going all the way back to the `airlines` table through $\mathrm{V}_3$ followed by forward tracing to the histogram through $\mathrm{V}_2$. Hence, expressing this interaction in provenance terms has the same structure (i.e., we backward trace and then forward trace) with the one we showed for linked brushing over `Delays`—even though $\mathrm{V}_2$ and $\mathrm{V}_3$ are more complicated than visualizing over the pre-materialized `Delays`. Finally, note that performance-wise SMOKE will perform provenance capture over the underlying joins which enables linked brushing to have similar performance with the one when tracing to materialized relations (e.g., `Delays`)—without having to explicitly pre-materialize relations, however, as we will see in Chapter 10.

### Cross-filtering

Cross-filtering is another interaction technique, in the multi-view linking class of interactions, and is typically used to explore correlated statistics across multiple visualization views [cro15]. In the common setup, each view is the result of an aggregation query over different combinations of input attributes (e.g., each view in Figure 7.1). Selecting marks in one view recomputes the aggregation queries over the subset of input records represented by the selection, and updates the views accordingly. Figure 7.13 illustrates a simple example

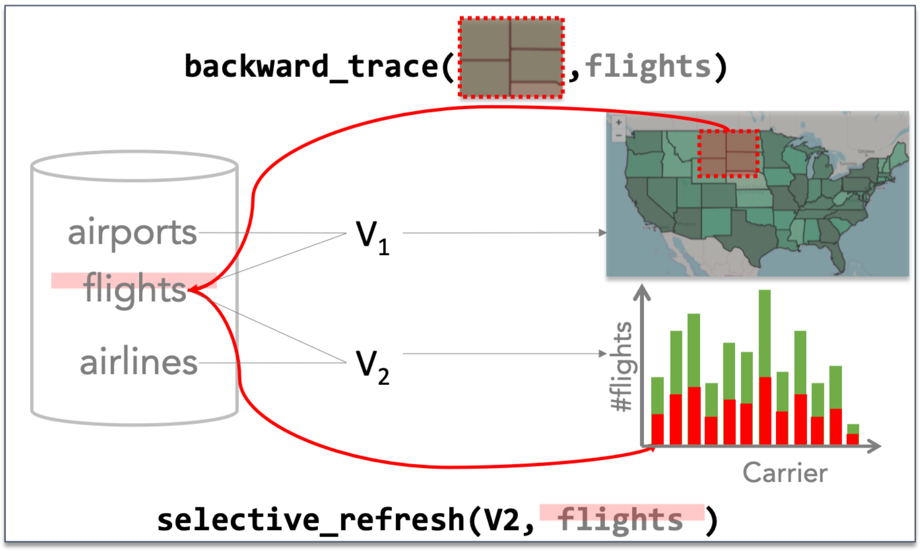


Figure 7.13: Crossfilter expressed using backward trace followed by selective refresh provenance statements.

where selecting a set of states updates the counts of flights per carrier. In Chapter 8, we will dive deeper into how to express and optimize such ubiquitous, data-intensive interactions. Here, we note that this interaction can be expressed in provenance terms by backward tracing the selected subset of states followed by a selective refresh provenance statement.

### Connection with Provenance

To summarize our findings, linked brushing can be expressed by backward tracing selected output marks to input records that contributed to the selected marks, followed by forward tracing to highlight marks in others (or even the same) views. Cross-filtering is expressed as backward tracing followed by selectively refreshing the other views (e.g., $V_1$ in Figure 7.13) over the provenance. The main difference is based on the forward tracing operation. In our examples, linked brushing traces the subset to the output marks, whereas cross-filtering recomputes the views for the output marks. Finally, by showing that these linking interactions are expressible in provenance terms, we showed that expressing such interactions in purely relational terms results in using logical provenance capture and query alternatives that have performance penalties and limited expressiveness.

### A Note On Semantics

To further highlight the importance of the provenance literature in the domain of interactive visualizations, we note that *selective refresh*, that we used to express crossfiltering, is a well-known and thoroughly studied provenance query construct. Selective refresh may not always update the same target outputs if the workflow contains a one-to-many operator followed by two non-monotonic aggregation operators [Ike12], in which case the refresh is unsafe. The notion of unsafe selective refresh, and recent techniques to address it [CLMR16], highlight the value of leveraging provenance to ensure correctness of interactive visualizations.

### A Note On Performance

Crossfilter is an important yet computationally expensive interaction technique. The visualization community has begun adopting dense [LJH13] and sparse [LKS13] data cubes to support cross-filtering at interactive speeds. Unfortunately, building such data structures requires considerable offline time–from minutes to hours on the ontime [Ont] flights dataset. This "cold-start" problem [BCHS17] makes it challenging for developers to rapidly build and test complex interactive visualizations, and makes it difficult to load a dataset in a visualization engine and immediately start cross-filtering. We will address this problem in Chapter 10. More specifically, we will show that it is possible to construct whole or partial data cubes for cross-filter provenance queries in interactive time. In addition, provenance metadata can be represented in efficient index data structures that accelerate backward and forward provenance tracing lookups. These forward and backward indexes are precisely the indexes to support incremental view updates on deletion.

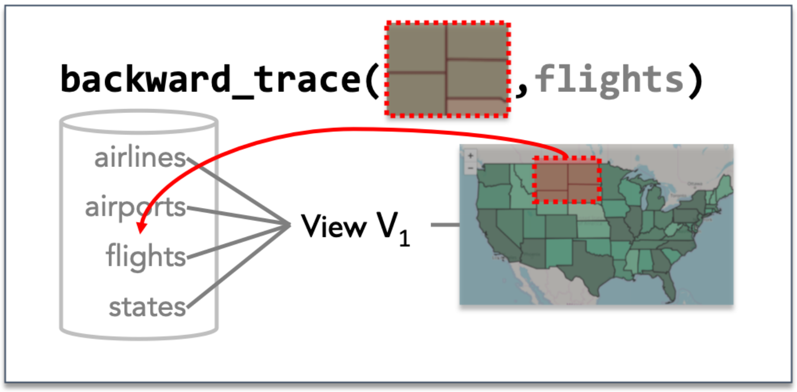


Figure 7.14: Selection interaction that shows the logical backward trace operation over $V_1$ to identify the subset of `flights` tuples that contribute to an interactive range selection.

## 7.5.3 Interactive Selections

One of the fundamental building blocks of visualization management systems is the ability to interactively reference visual marks by clicking, lassoing, or other types of selection operations [Tuk77; SMWH17; Wil03], which are techniques in the Select class in the Yi et al. taxonomy that we discussed in Section 7.4. Although users interact with visual marks, the intention is typically to manipulate the underlying data represented by the visual marks rather than the marks themselves. To this end, visualization research has developed many techniques to invert selections in pixel space to declarative selection queries in the input data space [SMWH17; HAW08; DKR97; LRB+97; NS00].

The predominant forms of selection are item/group selection and range selection. Consider the map in Figure 7.14. Item and group selection may correspond to clicking on one or more states with the overall selection corresponding to a set of states. The primary intention of such item/group selections is to identify the input records associated with the selected states. Range selection may correspond to drawing a bounding box (e.g., dashed red box in Figure 7.14). This may be interpreted as group selection, where the set of states corresponds to the state polygons that intersect with the box. However, the intention may also be to translate the bounding box into a predicate over `lat,lon` attributes over the shapes polygons. The latter representation can be attractive because the selection can be further manipulated and relaxed to, say, add additional predicates (e.g., `adelay` > 5min), modify the predicate clauses (e.g., increase the `lon` range), or remove unnecessary clauses [HAW08].

### Connection With Provenance and Instrumentation

All of the above selection types are variants of backward tracing provenance operations, which identify input records that contribute to specified output records. Different backward tracing semantics and implementations correspond to the above selection semantics.

**Range Selection.** Visualization systems typically support range selection when the visualization workflow consists of rescaling data attributes to visual variables (e.g., `COUNT` to `y` pixel position). Since the scaling operations are typically invertible, it is simple to, say, rescale the coordinates of the bounding box from $\texttt{y}_{\texttt{min}}$ to $\texttt{y}_{\texttt{max}}$ to be in terms of `COUNT`. Provenance research generalizes this by computing the workflow's inverse function $V_i^{-1}()$. This can be done through weak inverse functions [WS97] or deriving provenance predicates from relational workflows [Ike12]. Note that inverse functions are a form of lazy provenance capture that we discussed in Chapter 3. Here the focus, however, is not on inverting relational operators but rather user-defined functions (e.g., inverting linear scaling and anti-aliasing). Fine-grained provenance capture over user-defined functions is not currently supported in SMOKE. Yet our instrumentation framework accounts both for extending support to user-defined functions as well as helps on the synthesis of inverse functions (e.g., by allowing access and manipulations of predicates in selections, access state internal to SMOKE that UDFs may be using, and accessing the description of individual operators in a plan). Overall, expressing range selections as backward trace helps extend our support to visualizations that perform complex data processing as well as rendering.

**Item and group selection.** Item and group selections aim to identify the specific input records that correspond to the user's selection in pixel space. Visualization systems typically implement this by annotating records as they flow through the visualization workflow so that the output is annotated with the input records [BOH11]. However, annotations [BCTV04; NKG$^+$17] are only one mechanism to answer fine-grained provenance queries. They can also be computed by evaluating the provenance predicates above, or by explicitly materializing input-to-output record dependency information as explicit index data structures when executing the visualization workflow, as we showed in Chapter 3. In fact, similarly to how we argued on linked brushing in Section 7.5.2, evaluating using provenance predicates is a form of lazy provenance capture (this time on relational operations) while annotations are a form of logical provenance capture, and both approaches come with performance penalties on backward tracing evaluation, as we showed experimentally in Chapter 3.

### A Note On Semantics

One subtle point is that provenance systems may support different types of provenance semantics, and visualization developers should be aware of this semantics. For instance, assume we select outputs of `SC` and want the corresponding airlines from the `airlines` relation. We typically only want the set of airlines, rather than the bag of every copy of the airlines that were used to derive the selection. In this case, visualization toolkits should demand "which-provenance" semantics, that we showed how to evaluate and optimize in Chapters 4 and 5, as opposed to general transformation provenance semantics, that we showed how to derive in Chapter 3, that return each airline record as many times as it contributes to the selected outputs.

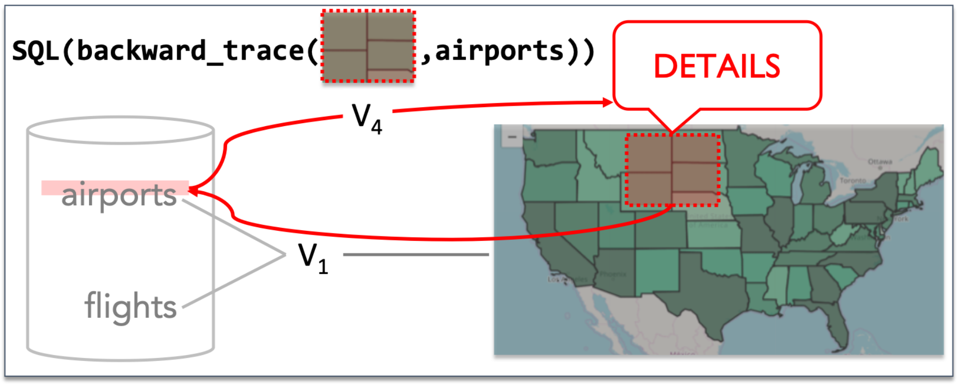


Figure 7.15: Logic over selections can be expressed as provenance consuming SQL queries to show information to users related to selections.

### 7.5.4 Logic Over Selections

A common use case once a user has performed a selection is to apply some logic over the selected data subsets (e.g., to show details, provide summarizations, drill down, or roll up). In fact, this logic over selection paradigm involves many techniques across classes in the Yi et al. taxonomy discussed in Section 7.4 because techniques in different classes are driven by selections. What changes, across techniques, is what logic is applied to the selection. To illustrate the logic over selection as a superclass of interaction classes, we will focus on tooltips, semantic zooming, and the more general class of details-on-demand which are popular examples of this paradigm.

Tooltips can render information (say, in a modal pop-up) that contains information about the provenance of the selected marks. For instance, when users select states in Figure 7.9, they may want to see additional attributes per state such as the average arrival and departure delays (i.e., `avg_adelay` and `avg_ddelay`, respectively). Furthermore, semantic zooming allows users to drill down into selections. For instance, if a user select states with a range selection on the map, the visualization may update to zoom into the range and show, say, detailed *city*-level breakdowns of counts of delayed flights. Finally, details-on-demand retrieve and further process user selections. For instance, when hovering over a state, the visualization may update to show a detailed list of airports operating in the state.

#### Connection With Provenance

These functionalities are often implemented as standalone features in visualization systems. However, they can be easily expressed as queries that take the backward trace of the user's selection as input. We illustrate this in Figure 7.15. The user selection in the visualization is traced back to input records. Then, a second visualization workflow $V_4$ computes statistics about the provenance and renders them as details. The primary distinction between the different examples above is the definition of $V_4$, which we illustrate in Figure 7.16.

**Examples.** The tooltip query `Z` traces the provenance of the user's `selected` states to the output of `SC` (i.e., `backward_trace(selected, SC)`), and returns the average departure

```
-- Tooltips
Z = SELECT   avg_adelay, avg_delay, state
    FROM     backward_trace(selected, SC);

Y = SELECT   SUM(cnt)
    FROM     backward_trace(selected, SC);

-- Details-on-demand
X = SELECT * FROM backward_trace(selected, airports);

-- Semantic zooming
W = SELECT   COUNT(*), city
    FROM     backward_trace(selected, flights) A1,
             backward_trace(selected, airports) A2
    WHER     A1.alid = A2.alid
    GROUP BY city;
```

Figure 7.16: Examples of tooltips, semantic zooming, details-on-demand

and arrival delay for each traced state. Another tooltip query `Y` again traces `selected` states to `SC`, yet this time computes a gross sum of all the delayed flights for the selection (i.e., in database terminology, it performs a roll-up). The details-on-demand shows two queries. `D` retrieves the list of airports within the selected states. Finally, the query `Z` performs the drill-down from state to city-level statistics, for the selected states. It does this by joining `flights` records and airports for the selected states, and re-computes the number of delays for each city. As a final note, all queries (i.e., `Z`, `Y`, `X`, and `W`) can be rendered to the user using visualization mappings on the way we have discussed in Section 7.3, and we omit their specification to avoid redundancy.

### A Note On Performance

Joins, such as the one in the query `Z` above, are common in visualizations. To avoid potentially expensive join execution costs, it is common practice for visualization systems and developers to denormalize relations ahead of visualization time. Static and interactive data visualizations are then implemented over the denormalized database.

However, denormalization is only one possible join optimization and comes with several costs. It introduces redundancy, is time- and space-consuming to construct, and in many cases not even required. Furthermore, this focus on denormalization is an example of violating physical data independence [Cod70] and impedes rapid visualization development. For instance, developers may spend considerable time writing application code to essentially denormalize `flights`$\bowtie$`airports` and compute the per-city count. Later, they may want to iterate on the visualization design and try showing, say, other statistics or grouping by `elevation`. However, they may be reluctant to incur the same engineering cost to try another design. This is because each design change implies the time- and space-consuming process of reconstructing the denormalized relation.

In contrast, expressing this logic in instrumentation, provenance, and relational terms enables rapid design iteration by offloading implementation and optimizations to the database engine. In fact, as we will see in Chapters 8 and 10, workflows composed in instrumentation, provenance, and relational terms can be optimized to ensure interactive response times by materializing efficient join indexes adaptively, partially denormalizing the database, and pre-computing statistics, among other optimizations.

## 7.6 Conclusions and Future Work

In this chapter, we introduced the core visualization and interaction constructs of iSQL. Furthermore, we showed that a purely relational approach, while adequate in expressing well-known interactions, leads to specifications that are hard to express and optimize. To address this problem, we showed how to express data-intensive interactions in simple provenance and instrumentation terms and push their optimization in our instrumentation-enabled engine. Given the low-level abstraction of iSQL augmented with provenance and instrumentation constructs, we believe the implementation of a compiler that transpiles existing high-level interaction grammars (e.g., Vega-lite [SMWH17]) to iSQL is an interesting problem with the goal to provide out-of-the-box optimizations to Web-scale applications.

# Chapter 8

# Crossfiltering and Incremental Cube Exploration

In the previous chapter, we discussed how to express common interactive visualization techniques in a blend of relational, provenance, and instrumentation terms. In this chapter, we show the performance benefits of expressing interactive visualizations in such terms, by optimizing one of the most data-intensive interaction techniques (i.e., crossfiltering).

## 8.1 Introduction

Crossfilter is an important interaction technique to help end users explore correlated statistics across multiple visualization views [cro15]. In the common setup, multiple group-by queries along different attributes of a dataset are each rendered as, say, bar charts. (Each bar chart corresponds to a visualization view.) When a user highlights a bar (or set of bars) in one view, the other views update to show the group-by results over only the subset that contributed to the highlighted bar (or bars). Consider the crossfilter interactive visualization example from Section 7.1 that we replicate here for convenience:

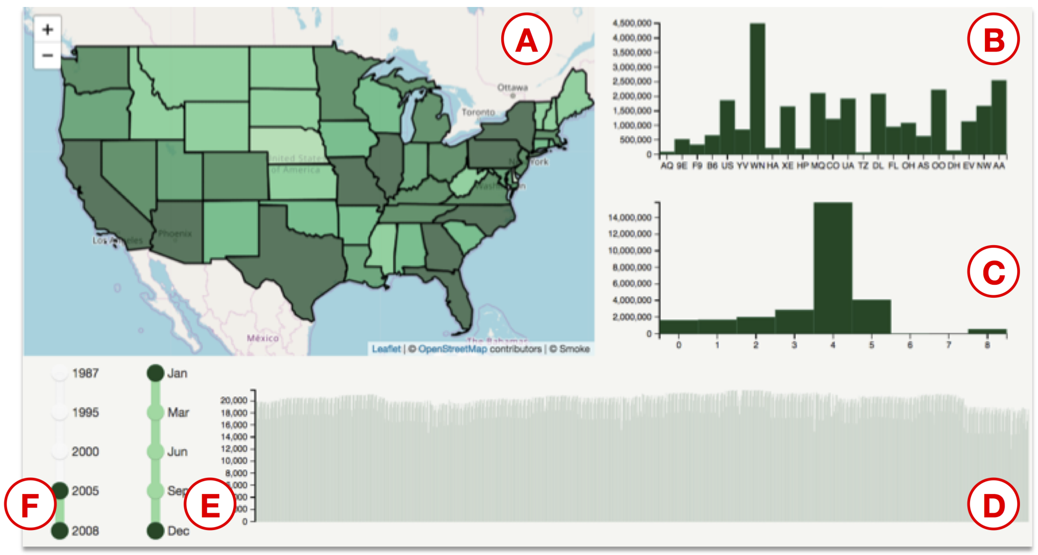


Figure 8.1: Crossfilter example.

**Example 13 (Flight delays exploration with crossfilter)** Figure 8.1 visualizes a breakdown of delayed flights [Ont] coupled with a crossfilter interaction technique [cro15]. Each chart corresponds to a count aggregation of delayed flights group by state A, by airline B, by hours of delay C, and by day D. The month E and year F sliders correspond to group by queries on year and month, respectively. An interactive range selection on the years triggers the re-execution of the aggregations, this time considering only records in the selected range [2005, 2008], and the update of charts to reflect only the aggregations in this time range. Beyond interactive selections of ranges, crossfilter can also be triggered by interactive selection of subsets of visual marks (e.g., bars in barcharts or regions in the map).

Since the views are fundamentally aggregation queries, recent research proposals construct variations of data cubes to accelerate the crossfilter interactions [LJH13; LKS13; PSSC17]. However, it can take minutes or hours to construct these data cubes. Such offline time is not available if a user has loaded a new dataset (e.g., into Tableau) and wants to explore using cross-filter as soon as possible. This has recently been referred to as the cold-start problem for interactive visualizations [BCHS17].

To address this problem, in this chapter we present provenance-based techniques that evaluate crossfilter interactions using provenance indexes built during the execution of initial group-by aggregations. As we will see experimentally, using provenance-based techniques allow us to respond, in most cases, interactively to crossfilter interactions without blocking users on offline construction of data cubes. For the few cases when crossfilter interactions will not be interactive, we introduce a technique that partially materializes a cube in-between crossfilter interactions without blocking users on their exploration.

Next, we first formalize the problem of crossfiltering (Section 8.2) and show in detail how it can be expressed using provenance queries (Section 8.3). Then, we present our provenance-based techniques as follows: In Section 8.4.1, we present a technique that responds to crossfilter interactions using lazy provenance query evaluation (i.e., using only SQL queries). In Section 8.4.2, we present a technique that evaluates crossfilter interactions using only backward indexes while in Section 8.4.3 we extend it to use both backward and forward indexes. In Section 8.4.4, we present our technique for partial cube materialization. Finally, we perform a cost analysis by means of memory that the different techniques utilize (Section 8.5) and present our experimental results (Section 8.6) to show that provenance-based techniques can perform on par with or better than data cubes on crossfilter interactions without blocking users on offline construction.

## 8.2 Problem Definition

Consider a relation $\mathrm{R}$ having attributes $\mathrm{A} = \{\mathrm{a}_1, \ldots, \mathrm{a}_\mathrm{n}\}$. Furthermore, consider a set of group-by aggregation queries $\mathcal{Q} = \{\mathtt{Q}_\mathrm{x} |$ `Qx = SELECT Dx, Fx(Mx) FROM T GROUP BY Dx`$\}$ where $\mathrm{D}_\mathrm{x}$ is a subset of the attributes A (often referred to as dimensions), $\mathrm{M}_\mathrm{x}$ is another subset of attributes $\mathrm{A}$ (often referred to as measures), and $\mathrm{F}_\mathrm{x}$ is a set of aggregate functions (e.g., SUM, COUNT, and AVG) to be computed for every group in $\mathrm{D}_\mathrm{x}$ over the measures $\mathrm{M}_\mathrm{x}$. The set $\mathcal{Q}$ constitutes the set of initial views for which we seek to support crossfiltering functionality.

A crossfilter interaction can be triggered by the interactive selection of output groups of $\mathcal{Q}$. Such interactive selections can happen in many ways in a visualization. Here, we discuss these selections based on single and multi-view brushes: a single-view brush selects a subset of output groups in a single view while a multi-view brush is the set of single-view brushes across views. More precisely:

**Single View Brush.** Let $\mathrm{Q_x(D)}$ denote the result of $\mathrm{Q_x}$ when applied in database D. Without loss of generality, we will denote $\mathrm{Q_x(D)}$ as $\mathrm{Q_x}$ (i.e., D remains fixed, updates are not allowed). We denote the selection effect of a single view brush on a view $\mathrm{Q_x} \in \mathcal{Q}$ as the subset $\mathrm{Q_x^{selected}} \in \mathrm{Q_x}$. Essentially, $\mathrm{Q_x^{selected}}$ is the subset of groups in the output $\mathrm{Q_x}$ selected by the brush. For instance, consider the map from our example:

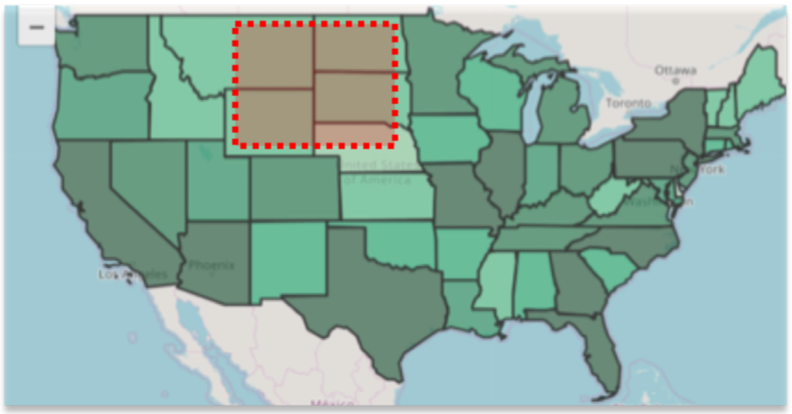


Figure 8.2: Brushing (red rectangle) selects a subsets of states.

The brush, in this case the red rectangle, selects a subset of states (which are groups in the output of the group-by state view in our example). This subset selection will then trigger the crossfiltering. In our example, we will need to update all the other views based only on the records that contributed to the selected states.

**Multi-View Brush.** Similarly to a single view brush we can also have multi-view brushes. These are constructed as the collection of single-view brushes. We denote this set as $\{\mathrm{Q_x^{selected}} \in \mathrm{Q_x}\}$. To account for a uniform representation of both single- and multi-view brushes note that a single-view brush $\mathrm{Q}_{\mathrm{x}}^{\mathrm{selected}}$ is essentially a singleton set of a multi-view brush. Hence, we can refer to the selected groups as $\{\mathrm{Q}_{\mathrm{x}}^{\mathrm{selected}} \in \mathrm{Q}_{\mathrm{x}}\}$ for both brush types.

**Crossfilter.** Given a single-view or multi-view brush $\{\mathrm{Q}_{\mathrm{x}}^{\mathrm{selected}} \in \mathrm{Q}_{\mathrm{x}}\}$ and the set $\mathcal{Q}$ of views defined as group-by aggregation queries, our goal set is to update all views in $\mathcal{Q}$ based only on the records that contributed to the selected groups. Depending on the interpretation of the selected outputs the crossfilter task at hand may differ. For instance, consider a multi-view brush that has selected a subset of states and another subset of carriers in our example dashboard. The interpretation of this brush may be that we want to update the other views based on the records that contributed both on the subset of states *and* the subset of carriers. Alternatively, we may want to update the other views based on the subset that contributed to states *or* carriers.

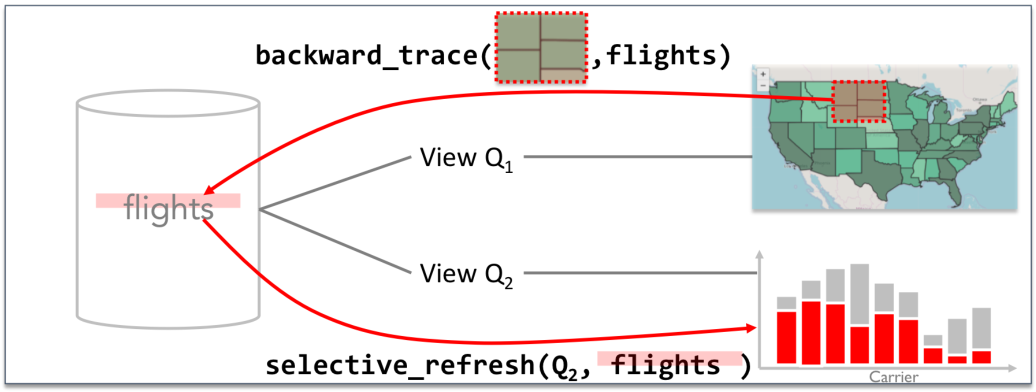


Figure 8.3: Expressing crossfilter interactions using provenance-based statements.

## 8.3 Expressing Crossfilter with Provenance Queries

As illustrated in the problem definition, evaluation of crossfilter interactions require us to a) identify the records that contributed to a selected set of output groups and b) refresh the other views based on these records. Our main idea, as we also discussed in Section 7.5.2, is that these two steps are directly expressible in provenance terms by a) backward tracing the selected outputs and b) selectively refreshing the other views based on the backward traced input partition of records.

To illustrate, consider our example as shown in Figure 8.3: A `backward_trace` statement identifies the partition of records in the input tables `flights` that contributed to the selected states. Then, a `selective_refresh` statement updates the distribution of the number of delayed flights per carrier based only the backward traced partition of the input. Note that the selective refresh statement is a `forward_trace` statement, that identifies the output groups in $\mathrm{Q}_2$ that each tuple in the input contributes to, followed by an `update` statement, that updates the counts of the forward traced output group.

## 8.4 Techniques

Having formalized the crossfilter problem and revisited how it can be expressed in provenance terms, we now proceed to show provenance-based techniques for its optimization.

### 8.4.1 Lazy

During the execution of each $\mathrm{Q}_\mathrm{x}$, LAZY executes the group-by aggregations without capturing provenance. Then, given a selection of a subset of outputs of $\mathrm{Q}_\mathrm{x}^{\text{selected}} \in \mathrm{Q}_\mathrm{x}$ from a single-view selection, LAZY supports crossfiltering by updating each $\mathrm{Q}_\mathrm{y} \in \mathcal{Q} \setminus \{\mathrm{Q}_\mathrm{x}\}$ through evaluating the group-by aggregation on the selected input records. In SQL terms, this can be expressed as shown in Section 8.4.1.

$$
\begin{aligned}
\mathrm{Q}'_\mathrm{y} \;=\; & \mathbf{SELECT}\ \mathtt{D_y},\ \mathtt{F_y(M_y)} \\
& \mathbf{FROM}\ \mathtt{R} \\
& \mathbf{WHERE}\ \mathrm{T.D_x}\ \mathbf{IN}\ \ \{\mathrm{o.D_x} | \mathrm{o} \in \mathrm{Q}^\mathrm{x}_{\mathtt{selected}}\} \\
& \mathbf{GROUP\ BY}\ \mathtt{D_y}
\end{aligned}
$$

Figure 8.4: Lazy approach for multi-output group selection.

Essentially, LAZY supports crossfiltering by performing lazy provenance capture to identify the partitions of the base relation that contributed to the selected outputs. Hence, for this particular case, LAZY reduces to first selecting the records that have the same group-by keys with the ones in $Q_x^{selected}$ and then executing the initial group-by queries only for the selected records. Since there may be multiple output groups selected in $Q_x^{selected}$, the selection of records that contributed to the selected output groups is a disjunction expressed with an **IN** statement, as shown in Figure 8.4. In our example, we can express the selection of input records based on the of selection of output states in the map as `state IN {'Montana', 'Wyoming', 'North Dakota', 'South Dakota', 'Nebraska'}`. Alternatively, we can express the selection with multiple OR statements (e.g., `state = 'Montana' OR state = 'Wyoming' OR state = 'North Dakota' OR state = 'South Dakota' OR state= 'Nebraska'`). Finally, note that the degenerate case, yet most typical case across crossfilter benchmarks, where only one output group has been selected (e.g., by clicking an individual output group), can be expressed with a single selection as shown in Figure 8.5.

```
Q'_y = SELECT D_y, F_y(M_y)
        FROM R
        WHERE T.D_x = o.D_x
        GROUP BY D_y
```

Figure 8.5: Lazy approach for single output group selection.

As we showed above, the semantics for single view selections are rather straightforward and typically result in disjunctive selections over input tables. The semantics for multi-view selections are open to interpretation. For instance, along with the selection of the states consider another selection of a carrier (e.g., `'AA'`). Even for this single carrier, there can be multiple interpretations for this multi-view selection. For instance, one may want to consider the input records with the state being one of the selected states `'Montana', 'Wyoming', 'North Dakota', 'South Dakota', 'Nebraska'` and carrier being the selected carrier `'AA'`. Yet another one, may want the records that have `(state IN {'Montana' OR 'Wyoming'} AND carrier = 'AA') OR (state IN {'North`

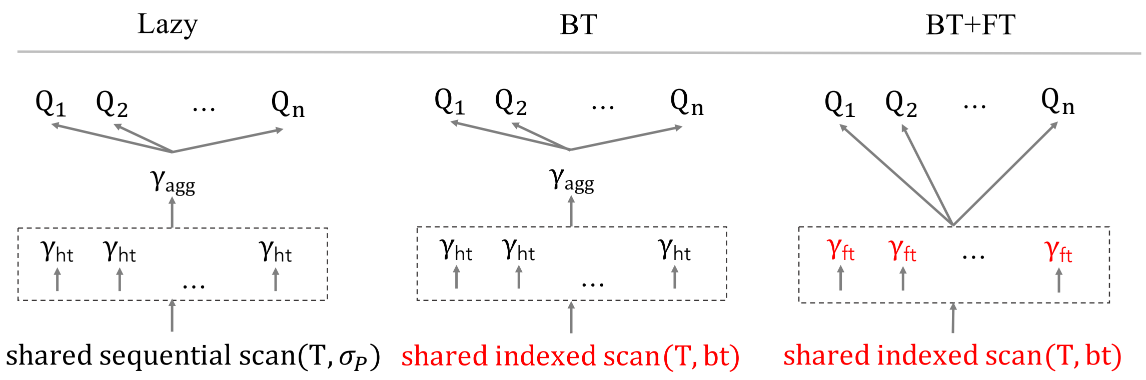

Lazy
BT
BT+FT
Q1 Q2 … Qn
γagg
γht γht … γht
γft γft … γft
shared sequential scan(T, σP)
shared indexed scan(T, bt)
shared indexed scan(T, bt)

Figure 8.6: Crossfilter evaluation techniques without using data cubes: (a) Lazy re-evaluates the group-by aggregation queries with a shared selection scan on the base table, (b) BT uses an index scan on the rids of the backward provenance index of $Q'_{\text{brushed}}$, (c) BT+FT performs updates using the forward indexes that connect each tuple in the base table to each output of aggregation query.

`Dakota', 'South Dakota', 'Nebraska'}).` Overall the actual SQL query for the LAZY in case of multi-view selections follows AND-OR semantics with the exact interpretation being application-dependent.

**Optimization.** Finally, to evaluate the updates on all $Q_x$, LAZY does not execute each update separately. Rather it uses a shared selection scan of the input relation to avoid multiple, expensive selection scans of the base relation.

As we noted in Chapter 3, LAZY approaches evaluate backward statements by typically rewriting them into equivalent selections over input tables. The problem with these approaches is that not every backward statement has an equivalent rewrite. For instance, consider in our setup for crossfilter the case where for each output $Q_x$ we do not store the group-by keys. In such cases, there is no way of rewriting the backward statements as selections. Furthermore, selections can be really expensive. In interactive visualizations benchmarks, for instance, in many common cases grouping happens on post-processed attributes. For instance, grouping by years or months which are extracted from timestamp attributes, by binned numerical attributes on various bin resolutions, or even by expensive

UDFs (e.g., classification algorithms) when turned into selections, as we described above, are very expensive to evaluate. Furthermore, the general AND-OR semantics may result in complicated selections that are also hard to evaluate during crossfiltering.

In such scenarios, instead of evaluating the selections, it is preferable to evaluate the selections using indexed scans that bypass the costs of selections. How to introduce indexed scans, however, if there are no indexes in the first place? This is the very essence behind the fine-grained provenance capture that we introduced in Chapter 3. Provenance capture on the group-by aggregation queries in our setup will generate the indexes required to evaluate the backward statements using indexed scans instead of expensive selections.

Next, we introduce two techniques (i.e., BT and BT+FT) that avoid the problems of LAZY by performing provenance capture during the execution of initial views to generate provenance indexes that speedup crossfilter interactions.

### 8.4.2 BT

During the execution of initial group-by aggregations, BT performs fine-grained provenance capture and associates each output group with a backward rid array. This rid array is a secondary index that can be used to identify the input records that contributed to this group during crossfiltering. This can be done by performing secondary indexed scans. In the case of a single-view selection of multiple groups, we can identify the input records by unioning backward rid arrays. Similarly, if we perform crossfiltering with AND semantics we can identify the input records by intersecting backward rid arrays. Also, in the case of AND-OR semantics, we can identify the input records by unioning and intersecting the rid arrays. Finally, note that, similarly to LAZY, BT also uses a shared scan, yet this time a shared indexed scan based on the rid arrays of the selected output groups, to perform crossfiltering.

### 8.4.3 BT+FT

Both LAZY and BT after they have identified the input records for the selected output groups they both need to perform group-by aggregations to evaluate the `selective refresh` statement, that we discussed in Section 8.3. In SMOKE, these are evaluated with hash-based group-by aggregations. However, building and probing hash tables are expensive operations for the interactive response time requirements of crossfilter applications. (Even if we implemented this functionality with sort-based group-by aggregations, again sorting partitions and grouping on them are expensive operations.) How can we avoid building and probing hash tables (or sorting and grouping) at crossfilter interaction time?

Consider the forward trace indexes for group-by aggregations. These indexes provide an important piece of information: what output group has each input tuple contributed to. This information provides exactly a perfect hashing between the input records and output groups and we can use it in place of hash tables to evaluate the `selective refresh` statements. Since they provide perfect hashing, there is no need to build or probe hash tables (or sort and group) but rather we can evaluate group-by aggregations by looking up outputs with array lookups provided through the forward rid arrays.

To this end, we introduce BT+FT as a technique that extends BT by using the forward indexes as perfect hashes and performs crossfiltering as shown in Figure 8.7: `agg_update()` updates the aggregation using our backward and forward indexes (e.g., for `COUNT(*)`, `agg_update` is $Q'_Z$`[fw[`$Q_Z$`][bw[i][j]]]++`). Furthermore, note the `remove_non_affected_groups` function at the end of the BT+FT algorithm in Figure 8.7. This function loops over the groups of each updated group-by and removes the groups that were not affected. In the case of `COUNT(*)` this is simply the groups that have a zero count. For other aggregates, like `SUM`, we need to track which groups were updated within the `agg_update` functions. However, in many interactive visualizations it is important to maintain the groups even if they have a zero count (or a zero sum). In such cases, the `remove_non_affected_groups` can simply be ignored and the `agg_update` can

```
Input:  bw[][]      // backward index for $Q_x^{selected}$
        fw[][]      // forward indexes from each tuple
                    // to groups of each initial
                    // group-by aggregation
        $Q_1, \ldots, Q_n$      // outputs of initial views
Output: $Q'_1, \ldots, Q'_n$      // crossfiltered views
Init $Q'_1, \ldots, Q'_n$ using $Q_1, \ldots, Q_n$
for i = 0 to bw.size()
  for j = 0 to bw[i].size()
    for z = 0 to n
      agg_update($Q'_z$[fw[$Q_z$][bw[i][j]]])
remove_non_affected_groups($Q'_1, \ldots, Q'_n$)
```

Figure 8.7: Crossfilter using BT+FT.

perform the update without updating a state of what groups were updated. Our experiments in Section 8.6 report the latency of BT+FT including the time for this operation.

### 8.4.4 Combining Provenance with Cubes

The main problem with BT and BT+FT is that when the backward traced subset is large, then their performance can become non-interactive due to the many aggregations that they have to compute online. To address this problem, we can combine data cubes with our provenance-based approaches. Figure 8.8 shows our proposed algorithm.

Since the main problem is on output groups that depend on large input subsets, our main idea is to pre-materialize the results for these groups. To do so, our algorithm in Figure 8.8 first orders the output groups based on their group cardinality in descending order by placing them in a min-heap. (Note that the group cardinality is known either through the query if the group-by aggregates compute counts or simply by looking at the cardinality of backward rid arrays.) After sorting, we materialize aggregates starting from the groups with the largest group cardinalities. To do so, we use the BT+FT approach, but note that we could have also

```
Input:  Q   // executed group-by aggregations
        bt  // backward indexes ∀q ∈ Q
        thr // threshold on storage consumption
// Sort groups on bt size
bt_sorted = sort(Q, bt)

// push the largest bt from each each query in heap
// heap is sorted on the size of the list for this group
H = min_heap();
for(i = 0; i < bt_sorted.size(); ++i)
  H.push({bt_sorted[i][0], i, 0});

while(1){
  {d, i, s} = H.top()
  H.pop()
  res = btft(d)
  storage.store(res)
  if(bt_stored[i].size()!=s+1)
    H.push({bt_sorted[i][s+1], i, s+1})
  if(storage.used > thr || H.size()==0)
    break
}
```

Figure 8.8: Partial cube materialization.

used either LAZY or BT. The algorithm stops materializing when either a memory budget is about to get exceeded or we have materialized the aggregates for all groups. Also, note that the materialization is for individual group selection on single-view brushes. To respond to multi-group selections for either single or multi-view brushes we can use the same idea with the pre-materialization in Figure 8.8. However, this results in large materialization costs. Essentially, if we compute the aggregates for all possible combinations of individual groups the end result is a full data cube which, as we argued in Section 8.1, takes a lot of time to construct. However, since the aggregates that we typically need to compute for crossfiltering are algebraic or distributed and not holistic, this means that we can compute aggregates for multi-group selections based on the aggregates for individual groups that Figure 8.8 materializes. Extending our technique to materialize aggregates for multiple groups together, to account for holistic aggregates, and to change the ordering criterion for what groups to materialize first are interesting future work.

## 8.5 Memory Footprint

Before proceeding with our experimental analysis, we first analyze the memory footprint required by our algorithms. Note that the LAZY approach does not require any memory while the memory required by our partial cube materialization technique in Section 8.4.4 is always bounded by a user-provided budget $\mathrm{thr}$. This leaves us with analyzing how much is required for the backward and forward indexes for the BT+FT and BT approaches.

Following our notation from Section 8.2, let a relation $\mathrm{R}$ have attributes $\{\mathrm{D}_1, \mathrm{D}_2, \ldots, \mathrm{D}_{|\mathrm{D}|}, \mathrm{M}_1, \ldots, \mathrm{M}_{|\mathrm{M}|}\}$, where $\mathrm{D}_\mathrm{i}$ denotes the dimension $\mathrm{i}$ and $\mathrm{M}_\mathrm{j}$ denotes a measure $\mathrm{j}$. Also, let $|\mathrm{R}|$ denote the cardinality of $\mathrm{R}$ (i.e., the number of records in $\mathrm{R}$).

To support crossfilter over $\mathrm{R}$, we execute group-by aggregations over each $\mathrm{D}_\mathrm{i}$. Hence, the total memory is the sum of memory for provenance indexes of each group-by aggregation:

$$\texttt{Total Memory} = \sum_{\mathrm{i}=1}^{|\mathrm{D}|} \mathrm{bw}_\mathrm{i} + \mathrm{fw}_\mathrm{i}, \mathrm{i} \in [1, |\mathrm{D}|] \tag{1}$$

where $\mathrm{bw}_\mathrm{i}$ is the memory for the backward provenance index and $\mathrm{fw}_\mathrm{i}$ the memory required for the forward index. Since the queries are group-by aggregations we have that the backward index is an rid index and the forward index is an rid array. We model the memory consumption of the backward rid index and the forward rid array below:

**Memory for the forward index,** $\mathrm{fw}_\mathrm{i}$**.** We start with the forward index since this is the simple case. Recall from the semantics of the forward index that we need to keep the output rid for each input tuple. Hence, the rid array has size equal to the cardinality of the input relation R (i.e., for each tuple in the input we store the rid of the output group that it contributes to). This means that the size of the forward index is equal to the size of a word $\mathrm{w}$ (assuming that each rid fits in a word) times the cardinality of $\mathrm{R}$:

$$\mathrm{fw}_\mathrm{i} = \mathrm{w} \cdot |\mathrm{R}|, \forall \mathrm{i} \in [1, |\mathrm{D}|] \tag{2}$$

Note that here we assume that we know the cardinality of the input and we do not pay extra costs for reallocations, which is not the case for the backward index.

**Memory for the backward index,** $\mathrm{bw_i}$**.** To model the memory for backward index we need to take into account two cases. These are the cases where we know the number of input records that contributed to each output group of the group-by aggregation queries and the case that we don't know them.

For the former case, we don't care about possible reallocations during provenance capture. The overall memory consumption under this case is simply equal to the size of a word times the size of the cardinality of $\mathrm{R}$.

$$\text{(Case 1: No reallocations required) } \mathrm{bw_i} = \mathrm{w} \cdot |\mathrm{R}|, \forall \mathrm{i} \in [1, |\mathrm{D}|] \tag{3}$$

For the latter case, we need to model the reallocations in rid arrays of the backward rid index. Lets assume that an rid array has initial capacity $\mathrm{C}$. Upon adding rids to the array and exceeding the initial capacity $\mathrm{C}$, we increase the size of the vector by a growth factor $\mathrm{k}$. That means that, after $\mathrm{n}$ reallocations, we will have allocated:

$$\mathrm{C}, \mathrm{C} \cdot \mathrm{k}, \mathrm{C} \cdot \mathrm{k}^2, \mathrm{C} \cdot \mathrm{k}^3, \ldots, \mathrm{C} \cdot \mathrm{k}^{\mathrm{n}}$$

To model the memory required by $\mathrm{bw_i}$, what we are interested is the last term $\mathrm{C} \cdot \mathrm{k}^{\mathrm{n}}$. Also, assume that the group-by aggregation query outputs $\mathrm{M}$ groups and each group has input cardinality $\mathrm{o_m}$ (i.e., #input records that contributed to group $\mathrm{m}$). To find how many reallocations are required to fit $\mathrm{o_m}$ rids in an array, we need to find the minimum $\hat{\mathrm{n}}_{\mathrm{m}}$ s.t.:

$$\frac{\mathrm{C} \cdot \mathrm{k}^{\hat{\mathrm{n}}_{\mathrm{m}}}}{\mathrm{o_m}} \geq 1$$

As a result, the memory required for $\mathrm{bw_i}$ is the sum:

$$\text{(Case 2: Reallocations required) } \mathrm{bw_i} = \sum_{\mathrm{m}=0}^{\mathrm{M}} \mathrm{C} \cdot \mathrm{k}^{\hat{\mathrm{n}}_{\mathrm{m}}} \tag{4}$$

By substituting (2) and (3) in (1) we get the total memory if no reallocations are required:

$$\text{(Case 1) } \texttt{Total Memory} = 2 \cdot |\mathrm{D}| \cdot \mathrm{w} \cdot |\mathrm{R}| \tag{5}$$

By substituting (2) and (4) in (1) we get the total memory if reallocations are required:

$$\text{(Case 2) } \texttt{Total Memory} = |\mathrm{D}| \cdot \mathrm{w} \cdot |\mathrm{R}| + \sum_{\mathrm{i}=1}^{|\mathrm{D}|} \sum_{\mathrm{m}=0}^{\mathrm{M}} \mathrm{C} \cdot \mathrm{k}^{\hat{\mathrm{n}}_\mathrm{m}} \tag{6}$$

## 8.6 Experimental Settings

**Setup.** Following previous studies [LJH13; LKS13; PSSC17], we used the Ontime dataset and four group-by COUNT aggregations on <lat, lon> (65,536 bins), <date> (7,762 bins), <departure delay> (8 bins), and <carrier> (29 bins); only 8,100 bins have non-zero counts because <lat, lon> is sparse. Each group-by query corresponds to one output view. This setup favors cube construction because it involves only four views and coarse-grain binning on spatiotemporal dimensions (which decreases the size of cubes and increases group cardinalities). To trigger crossfilter interactions we select every possible group from every group-by output view.

**Techniques.** We compare the following: **LAZY** uses lazy provenance capture and re-executes the group-by queries on the provenance subset. **BT** uses SMOKE to capture backward provenance indexes but re-runs the group-by queries (which requires re-building group-by hash tables). **BT+FT** also captures forward provenance indexes that map input records to the output bars that they contribute to, which can be used to incrementally update the visualization bars without re-building group-by hash tables. We compare our techniques with **DATA CUBE** construction. We first ran IMMENS [LJH13], NANOCUBES [LKS13], and HASHEDCUBES [PSSC17] to construct the data cubes. However, IMMENS and NANOCUBES did not finish within 30 minutes, while HASHEDCUBES required 4 minutes. For this reason, we implemented a custom partial cube construction based on our group-by aggregation push-down optimization that took 1.6 minutes to construct. This construction resembles the low

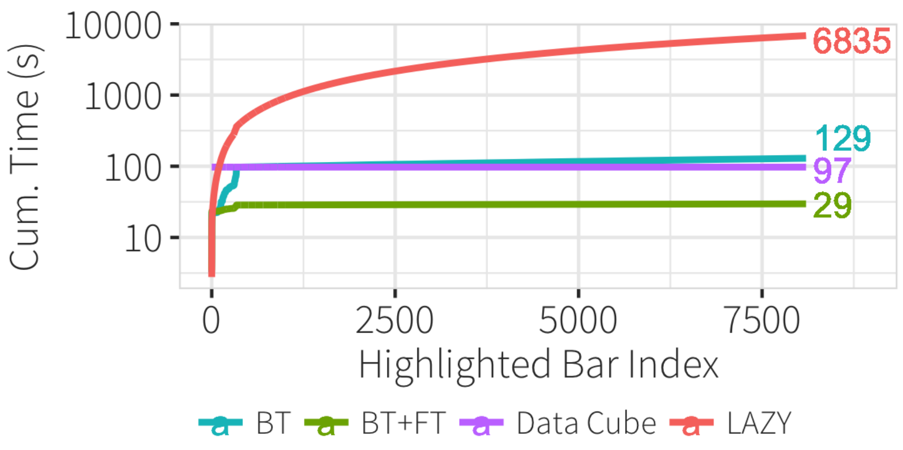


Figure 8.9: Cumulative latency of different crossfiltering techniques. BT+FT outperforms all approaches with the total time to perform the initial group-by aggregates, track provenance, and evaluate all interactions being thirty seconds.

dimensional cube decomposition described by IMMENS but using the sparse encoding recommended by NANOCUBES. Finally, we refer to our technique in Section 8.4.4, that blends data cubes with provenance for incremental cube exploration, as **PARTIALCUBE+BTFT**.

**Platforms.** We ran experiments on a server-class machine running Ubuntu 14.04, and having a 64GiB 2133MHz DDR4 memory (caches sizes 32KiB L1d, 32KiB L1i, 256KiB L2, and 10MiB L3) and 3.1GHz Intel Xeon E5-1607 v4 processor.

## 8.7 Experimental Results

We start by evaluating the performance of LAZY, BT, BT+FT, and DATA CUBE. Figures 8.9 and 8.10 report the individual and cumulative latencies to highlight each and every bar, respectively, per our experimental settings.

**Comparison of LAZY, BT, BT+FT, and DATA CUBE.** We make four main observations. First, we observe that BT outperforms LAZY by leveraging the backward index to avoid table scans; BT+FT outperforms BT because the forward index lets SMOKE directly update the associated visualization bars without the need to re-build group-by hash tables; and,

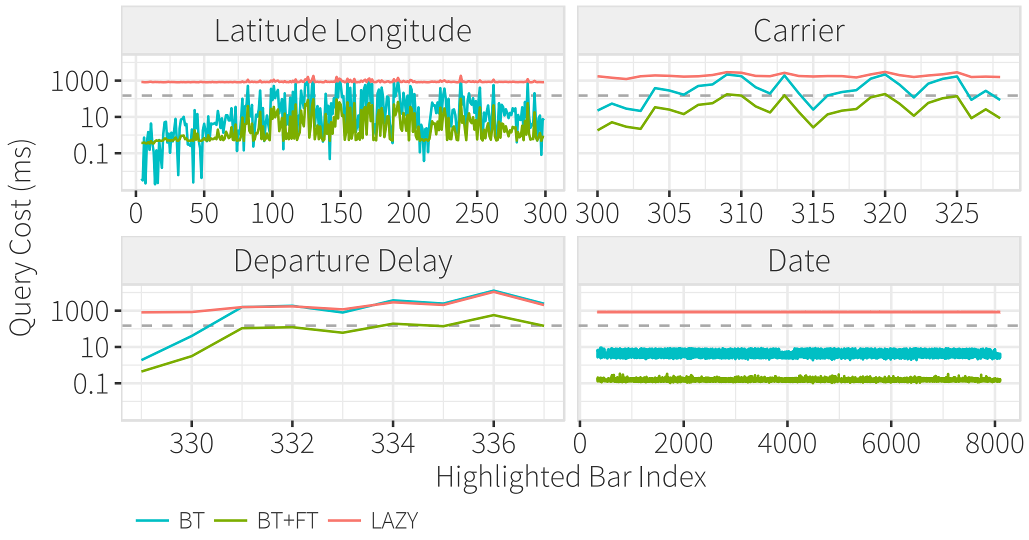


Figure 8.10: Latency for each crossfilter interaction. Dashed lines correspond to 150ms interaction layer. BT+FT performs under the 150ms interaction layer for all 8,100 but 5 interactions, with interactions on the spatiotemporal dimensions to be <10ms. Data Cube has instantaneous response time and we do not plot it.

although the DATA CUBE response time is near-instantaneous, the cube construction cost is considerable and BT+FT is able to complete the benchmark before the cube is constructed (Figure 8.9). Second, BT+FT performs best ($<$ 10ms) when group-by queries output many groups (e.g., lat/lon and date) because then each group's backward provenance is substantially small. This suggests that provenance can complement cases when data cubes are expensive (e.g., when a cube dimension contains many bins) by computing the results online, as we will discuss with our results using PARTIALCUBE+BTFT. Third, Figure 8.10 shows that BT+FT responds within $<$ 150ms (dotted line) for all but five bars, whose provenance depends on a large subset of the input records (>10% selectivity; >13M records). Fourth, the capture overhead for BT+FT and BT on the initial group-by queries are relatively low ($< 2\times$ using SMOKE-I). We expect optimizations that use parallelization, sampling, and deferred provenance capture to reduce crossfilter latencies even further.

**Performance and semantics of PARTIALCUBE+BTFT.** Furthermore, we evaluated the performance of PARTIALCUBE+BTFT on the same benchmark. Recall that for PARTIAL-CUBE+BTFT we need to set a memory budget. To understand the overall implications of PARTIALCUBE+BTFT we have set the memory budget to two extremes: unbounded and 0. Regarding the former, note that the cumulative latency of PARTIALCUBE+BTFT is the same with the one of BT+FT (i.e., every crossfilter interaction will be served by performing BT+FT). Regarding the latter, the overall latency of PARTIALCUBE+BTFT to pre-materialize the results of possible interactions is ~33 seconds. Essentially, this results in performing BT+FT for every possible bar plus the time required to perform the ordering of indexes (i.e., ~4 seconds). Now, note that the latency for crossfiltering using PARTIALCUBE+BTFT depends on whether or not the crossfiltering results have been pre-materialized. If that's the case, then the latency is simply the latency of fetching the materialized results (i.e., ~0ms). Otherwise, the latency is equivalent to the latency of BT+FT. To this end, the worst performance for crossfiltering is the case where the user always selects a bar for which crossfiltering results have not been materialized yet. In this case, the performance is equivalent to BT+FT. This is a highly unlikely event given that within 29 seconds a user needs to have performed 8,100 crossfiltering interactions and understood the results of every possible such interaction. To conclude, the best case performance for crossfiltering using PARTIALCUBE+BTFT is equivalent to DATA CUBE and the worst performance is equivalent to BT+FT.

So far, we have only experimented with single bar selections which is common in crossfiltering benchmark. However, as we noted in Section 8.2, crossfiltering can be triggered by multiple selections through either single-view or multi-view brushes. Furthermore, we noted that such crossfilter interactions carry AND-OR selection semantics. Here we briefly discuss our results on worst and best cases under AND and OR semantics.

**AND semantics.** Regarding AND semantics, the worst case performance comes from selecting the output groups with highest input cardinalities from each output view. The latencies of LAZY, BT, and BT+FT for this case in our experiments are ~16s, ~16s, and

~0.8s, respectively. These results l highlight the overall performance benefits of BT+FT. Yet, note that 0.8s may still be regarded as a non-interactive response time. By using PARTIALCUBE+BTFT to materialize groups in decreasing group cardinality order, we can respond to this query near-instantaneously even after the materialization of the first five groups in this order. The best case performance under AND semantics (assuming multi-view selection) comes from selecting the groups with the lowest cardinality from each output view. In this case, the latencies of LAZY, BT, and BT+FT are ~3.6s, ~0.025s, and ~0.002ms, respectively, and further highlight the benefits of our provenance-based techniques.

**OR semantics.** Regarding OR semantics, note that the worst possible performance comes from when the user selects all possible bars from every (or even all bars from a single) output view. This case results in re-execution of the initial group-by queries. To account for more meaningful worst case experiments, we sum up the amount of time required to highlight every bar in each output view. For the temporal view, the total latencies required by LAZY, BT, and BT+FT are 6,475.3s, 32s, and 1s, respectively. For the map view, the total latencies required are 262.3s, 31.5s, and 2.3s for LAZY, BT, and BT+FT, respectively. These results highlight the performance benefits of our provenance-based techniques for output views that contain multiple groups each with low group cardinality, even under OR semantics. For the departure delay view,the latencies are 25.5s, 24.3s, and 1.3s for LAZY, BT, and BT, respectively. Finally, for the unique carrier view the latencies are 56.3s, 17.7s, and 1.5s, for LAZY, BT, and BT, respectively. Our results on departure delay and unique carrier highlight the problem of BT that, given groups of high group cardinality, is affected by the hash table building, and further shows the benefits of BT+FT that avoids this problem. Finally, note that best case performance for OR semantics (assuming multi-view selections) is when users select a single group from every output view. In this case, the performance of the compared techniques is similar to the best case performance under AND semantics.

Furthermore, note that arbitrary compositions of AND-OR crossfilter semantics is also possible. While not covered by our experiments since we have already covered AND and OR semantics independently, we reiterate our general observation on the crossfiltering performance: whenever the cardinality of the input partition that will be used by the current crossfilter interaction is low ( <10%-20% in our experiments), then BT is significantly faster than alternative approaches and remains interactive. For higher cardinalities, we believe that extensions to our partial cube materialization strategy to account for AND-OR semantics as well as the incorporation of query execution optimizations (e.g., vectorization) can further improve the performance of our techniques on such complicated crossfilter interactions.

Finally, an important note is that our observations are over a single aggregation function (i.e., COUNT). In general, different aggregation functions may have different evaluation complexities As a result, our observation on the correlation of our techniques with input group cardinalities should be reconsidered in case aggregation functions under consideration have significantly different complexities.

## 8.8 Conclusions

In this chapter, we addressed the problem of evaluating crossfilter interactions when there is no sufficient time or space to compute and materialize data cubes offline, respectively. To do so, we showed how to express crossfiltering using provenance queries which allowed to devise provenance-based techniques for their evaluation. Our experimental results show that instrumentation-enabled (and by extension provenance-enabled) database engines are powerful enough to support one of the most data-intensive interactions without sacrificing performance. Going forth, we believe there is ample space for combining provenance with data cubes. For instance, the order that we used in PARTIALCUBE+BTFT for materialization only considers group cardinalities. Having control of the interaction space, however, may allow us to drive what needs to be materialized (e.g., brush interactions may select only consecutive bars or users have zoom in the map and focus only a small subset of the output groups). Furthermore, we believe combining provenance-based evaluation techniques with sampling, vectorization, and other query execution optimization techniques can further decrease the crossfilter interaction time and increase the user engagement.

# Chapter 9

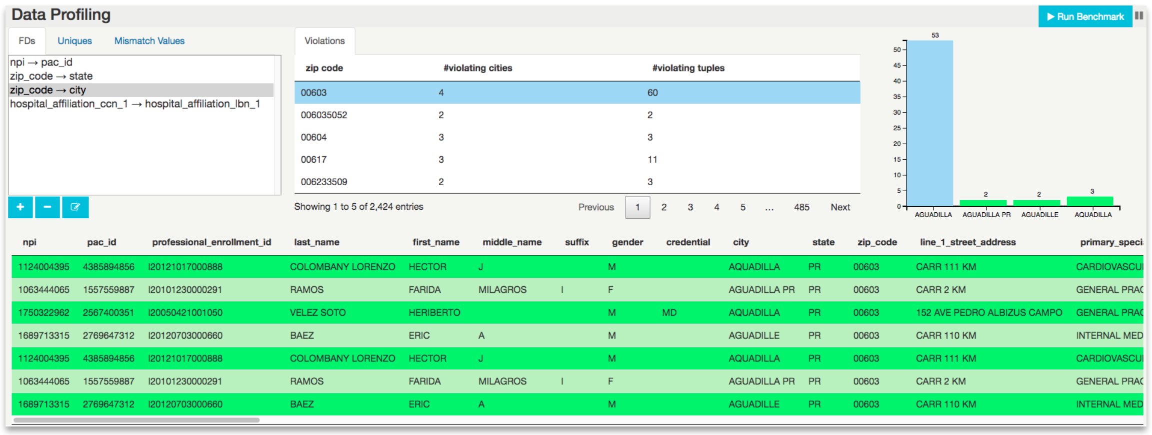
# Interactive Data Profiling

In this chapter, we continue our discussion over applications domains that instrumentation-enabled database engines can facilitate their expressiveness and optimization by diving into the domain of interactive data profiling.

## 9.1 Introduction

Data profiling studies the statistics and quality of datasets (e.g., constraint checking; data type extraction; or key identification) while interactive data profiling [Nau14] allows users to interactively profile and examine the reasons for these results. Recent systems include extensible data profiling platforms (e.g., METANOME [PBF+15]), data wrangling and cleaning tools (e.g., Wrangler [KPHH11], Profiler [KPP+12], and NADEEF [EEI+13]), and user-guided functional dependency (FD) miners (e.g., UGUIDE [TBEO+17]).

In such systems, profiling results can be viewed as the results of data-processing workflows, and the interactive profiling functionality corresponds to inspecting raw (or summarized) inputs that contributed to these output profiling results. For instance, UGUIDE mines datasets for FDs and presents violations of candidate FDs to the user to validate. Similarly, data cleaning applications typically render summary statistics as bar charts or heat maps [KPP+12] that the user can interactively inspect.



npi → pac_id
zip_code → state
zip_code → city
hospital_affiliation_ccn_1 → hospital_affiliation_lbn_1

| zip code | #violating cities | #violating tuples |
|---|---|---|
| 00603 | 4 | 60 |
| 006035052 | 2 | 2 |
| 00604 | 3 | 3 |
| 00617 | 3 | 11 |
| 006233509 | 2 | 3 |

Showing 1 to 5 of 2,424 entries

| npi | pac_id | professional_enrollment_id | last_name | first_name | middle_name | suffix | gender | credential | city | state | zip_code | line_1_street_address | primary_speci |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1124004395 | 4385894856 | I20121017000888 | COLOMBANY LORENZO | HECTOR | J |  | M |  | AQUADILLA | PR | 00603 | CARR 111 KM | CARDIOVASCU |
| 1063444065 | 1557559887 | I20101230000291 | RAMOS | FARIDA | MILAGROS | I | F |  | AGUADILLA PR | PR | 00603 | CARR 2 KM | GENERAL PRAC |
| 1750322962 | 2567400351 | I20050421001050 | VELEZ SOTO | HERIBERTO |  |  | M | MD | AQUADILLA | PR | 00603 | 152 AVE PEDRO ALBIZUS CAMPO | GENERAL PRAC |
| 1689713315 | 2769647312 | I20120703000660 | BAEZ | ERIC | A |  | M |  | AGUADILLE | PR | 00603 | CARR 110 KM | INTERNAL MED |
| 1124004395 | 4385894856 | I20121017000888 | COLOMBANY LORENZO | HECTOR | J |  | M |  | AQUADILLA | PR | 00603 | CARR 111 KM | CARDIOVASCU |
| 1063444065 | 1557559887 | I20101230000291 | RAMOS | FARIDA | MILAGROS | I | F |  | AGUADILLA PR | PR | 00603 | CARR 2 KM | GENERAL PRAC |
| 1689713315 | 2769647312 | I20120703000660 | BAEZ | ERIC | A |  | M |  | AGUADILLE | PR | 00603 | CARR 110 KM | INTERNAL MED |

Figure 9.1: Interactive data profiling interface.

To further demonstrate this functionality, Figure 9.1 shows an interface that we have built as a SMOKE client for interactive data profiling purposes. At the top, the left panel is used to select an FD, uniqueness, or mismatched value constraint to check—which are the main data profiling tasks we consider in this section. Here the user has selected the FD `zip_code→city`. The middle panel renders a summary of the check results in terms of the zipcodes that have more than one city value. Selecting a violation updates the right panel, which shows the distribution of city values for that zipcode, and the bottom panel, which renders a table with the individual records that contributed to the violations. This table can be further restricted by selecting a subset of city values in the top right bar chart.

To provide this functionality, in this section we cast the evaluation of data profiling tasks (i.e., interactive exploration of FD, uniqueness, and mismatch checks) and the exploration of data profiling results as operations involving backward and forward provenance queries.

More specifically, our contributions in this chapter are as follows. First, we introduce provenance-based techniques for the evaluation and interactive exploration of FD (Section 9.2), uniqueness (Section 9.2), and mismatch (Section 9.4) checks. Then, we show experimentally that our techniques improve on alternative state-of-the-art, hand-written implementations (Section 9.5). These results highlight the power of instrumentation-(and by extension provenance-)enabled database engines, and suggest that data profiling tools can express their logic declaratively in provenance terms all while improving their performance.

## 9.2 Evaluating Functional Dependencies

In this section, we show how to evaluate FDs and explore violations of them interactively. The main sketch of our approaches is to express FD violation checks as relational workflows and track provenance during their execution. Hence, each violation is connected with the records responsible for the violation through the underlying provenance graph. As such, users can interactively explore the violation by, say, inspecting the records responsible for the violation by tracing backward on the provenance graph from the output violation.

Next, we present two approaches based on the above sketch, namely, **CD** and **UG**. As we will see, in the second approach we will not just use provenance for interactive exploring FD violations but rather also for evaluating FD violation checks. This further highlights the importance of provenance in the interactive profiling domain. To ease of our discussion assume that we want to find and explore the violations for the FD $\mathtt{A}{\rightarrow}\mathtt{B}$ over a table $\mathrm{T}$.

**CD**. Our first approach identifies violations of the FD $\mathtt{A}{\rightarrow}\mathtt{B}$ over the table $\mathrm{T}$ with the following query $\mathrm{Q}_{\mathrm{CD}}$, which is the typical way to evaluate FDs of this form in SQL terms:

```
Q_CD  =  SELECT A
          FROM T
          GROUP BY A
          HAVING COUNT(DISTINCT B) > 1
```

Figure 9.2: Query for extracting violations on a FD $\mathtt{A}{\rightarrow}\mathtt{B}$. By tracking provenance on this query we can connect the violating $\mathrm{A}$ values with the records responsible for the violation.

$\mathrm{Q}_{\mathrm{CD}}$ outputs the distinct values $\mathrm{a} \in \mathrm{T.A}$ that violate the FD. Now, consider tracking backward provenance on $\mathrm{Q}_{\mathrm{CD}}$. This results in connecting the input records $\{\mathrm{t} \in \mathrm{T} \mid \mathrm{t.A} = \mathrm{a}\}$ with each violating value $\mathrm{a}$. Using this provenance information, we can enable users to inspect the violating input records (e.g., with backward provenance queries), collect statistics over violations (e.g., using provenance consuming SQL queries), or prompt users for cleaning purposes to overall expose interactive data profiling capabilities.

**UG**. The second approach (**UG**) is based on an optimization in UGUIDE's METANOME-based implementation. Through correspondence with the authors, it turns out that the implementation effectively simulates provenance indexes, and thus we describe it in provenance terms. We first evaluate the following query for the attributes in the FD attr$\in \{A, B\}$ and at the same time we capture both backward and forward provenance:

```
Q_ug^attr=SELECT attr
          FROM T
          GROUP BY attr
          HAVING COUNT(1) > 1
```

We then backward trace each $a \in Q_{ug}^{A}$ to the input $T$, and forward trace each provenance record to $Q_{ug}^{B}$. If more than one distinct $b$ values are in the forward traced output, then the FD is violated, and the provenance indexes connect the violation with the tuples that contributed to the violation, similarly to the result of the CD approach.

As we will see in our experiments, CD is faster than UG for the evaluation of individual FDs. However, we note that UG is typically faster than CD for batch evaluation of FDs.

## 9.3 Evaluating Uniqueness

To check uniqueness for an attribute U, we simply execute $Q_{ug}^{U}$ from above to identify values in U that are not unique. The backward provenance for an output record corresponds to the input records that contribute to the uniqueness violation. This also illustrates how provenance from the same query can be shared across data profiling algorithms.

Another similar, yet more complicated, uniqueness check profiling task is to identify the unique values of an attribute U for a given value of a different attribute V. This task is equivalent to crossfiltering that we presented in Chapter 8. While our experiments with crossfiltering in Section 8.6 focused only on group-by aggregation over 4 attributes, the typical case for profiling is on wide tables with many attributes. Hence, in our experiments in this section, we will show the performance of our provenance-based techniques for crossfiltering over wide tables for uniqueness check purposes.

## 9.4 Evaluating Mismatches

Finally, mismatches are expressed with selections over the input table that should evaluate to true but do not. Such constraints are commonly used to identify domain or type violations, and they can be quite expensive depending on the complexity of the selection predicate. By tracking provenance over mismatch checks, we can allow users to explore mismatches without re-evaluating expensive predicates. Essentially, this is similar to our results in Section 3.7 with provenance queries when base queries involve expensive selections. As we noted in Section 3.7, LAZY approaches (i.e., provenance querying by re-evaluating expensive predicates) are more expensive than our SMOKE-L technique. This is because SMOKE-L avoids the expensive selection scans of LAZY through provenance-based indexed scans.

## 9.5 Experiments

Our experiments seek to show evidence that provenance-based techniques allow developers of interactive data profiling applications to express their logic declaratively without loss in performance. To do so, we evaluate our techniques on evaluating functional dependency and uniqueness checks against state-of-the-art, hand-written alternatives. (For mismatches we have already shown experimental evidence in Section 3.7 and we omit further evaluation.)

**Dataset.** For our experiments in this chapter we use the Physician [Phy] dataset (2.2m tuples, 0.6GB, 41 attributes) which was used in the Holoclean [RCIR17] paper for data cleaning purposes. We use this dataset as it has known functional dependencies that are violated, as specified in Holoclean [RCIR17], and because it has many attributes, which is important for our experiments on uniqueness checks.

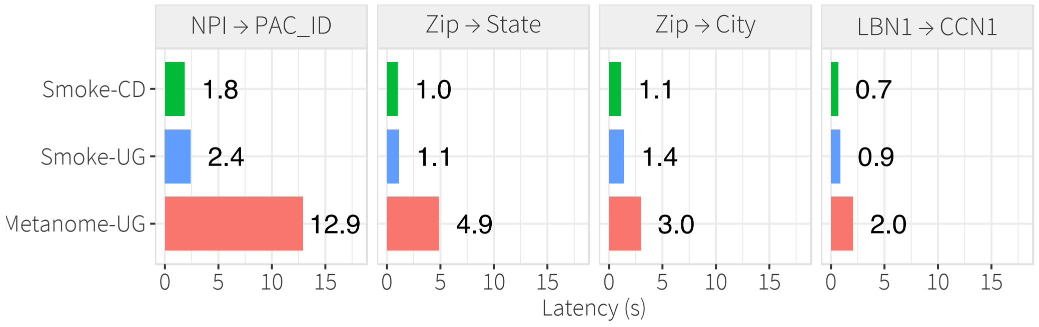


Figure 9.3: Latency of different approaches for FD violation evaluation and bipartite graph construction. SMOKE-CD is the minimal overall. METANOME-UG is affected by virtual function calls for provenance capture, the overheads of JVM, and its data model.

**Compared techniques and setup on FD checks.** For the evaluation of FDs, we compare SMOKE that implements both of our **CD** and **UG** approaches with UGUIDE that implements the **UG** one in METANOME. We refer to our SMOKE techniques as SMOKE-CD and SMOKE-UG and to the UGUIDE one as METANOME-UG. The comparison is on absolute latency for the evaluation of four FDs over the Physician dataset (i.e., `NPI→PAC_ID`, `Zip→State`, `Zip→City`, and `LBN1→CCN1`).

**Compared techniques and setup on uniqueness checks.** For the evaluation of uniqueness checks, we compare the LAZY, BT, BT+FT, and DATA CUBE approaches that we presented for crossfiltering. Our results here aim to show the performance of these techniques over many attributes which, while unconventional for interactive data visualization purposes, it is important for data profiling purposes. For this experiment, we use only 20 out of the overall 41 attributes in the physician dataset because the DATA CUBE approach has quadratic on the number of attributes complexity, rendering it prohibitively expensive for 41 attributes.

## FD Evaluation

Figure 9.3 compares the latency of the FD evaluation techniques using the four FDs over the Physician dataset, as we discussed in our setup. Overall, SMOKE-UG outperforms METANOME-UG by $2-6\times$ while the simpler SMOKE-CD approach outperforms both

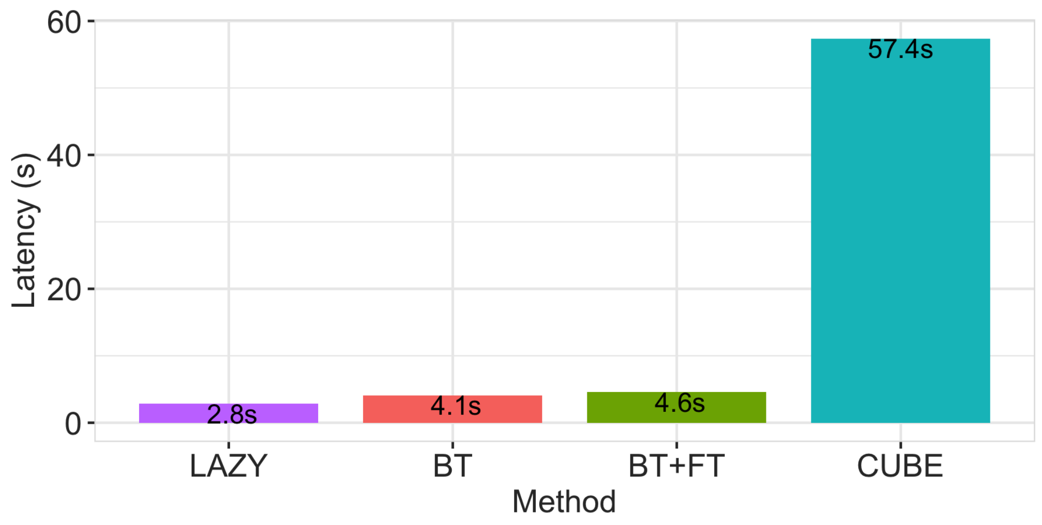


Figure 9.4: Latency of different approaches for uniqueness checks.

approaches. Both SMOKE capture overheads ($< 1.2\times$ overhead) are consistent with our microbenchmarks in Section 3.7. There are several reasons why SMOKE-UG outperforms METANOME-UG. METANOME-UG incurs virtual function call costs when constructing its version of provenance indexes ($> 2\times$ overhead on $\mathtt{Q}_{\text{ug}}^{\text{attr}}$ that we implemented in UGUIDE), as well as general JVM overhead even after a warm-up phase to enable JIT optimizations. Furthermore, METANOME-UG models all attribute types as strings, which slows uniqueness checks for integer data types such as `NPI`. To account for a fair comparison, the other three FDs are over string attributes (zip is a string).

## Uniqueness Evaluation

Figure 9.4 shows the performance of different techniques to support uniqueness checks. As we noted in Section 9.3 the different techniques are the same with the ones we use for crossfiltering, and the results are similar with the ones we showed in Section 8.6. A major difference, however, is that the DATA CUBE approach in this setting takes substantially more time than the other approaches. This is because DATA CUBE has quadratic complexity on the number of attributes whereas the complexity of the other techniques is linear.

## 9.6 Conclusions

In this chapter, we drew the connections between instrumentation-enabled (and by extension provenance-enabled) engines and the domain of interactive data profiling. Our experimental results show evidence that provenance capture and querying as implemented in SMOKE enable interactive data profiling applications developers to express their techniques declaratively in provenance terms and meet or even gain in performance compared to hand written alternatives. Going forth, we believe there are many directions in the intersection of data profiling and instrumentation engines worth of further exploration. For instance, the techniques that we introduced in this chapter cover only a small fraction of possible profiling techniques without covering, say, the general class of conditional dependencies. (Refer to [Nau14] for a classification of data profiling tasks.). Furthermore, while in Chapter 6 we denoted how more attributes can be added and removed from plans and how instrumentors can piggyback computations within plans, these only provide the mechanisms that data profiling tasks; such as piggybacking cardinality; data distribution; and small materialized aggregates; can build upon. Putting them into practice per application domain is interesting future work.

# Chapter 10

# Physical Database Design

Our instrumentation-based techniques in support of crossfiltering (Chapter 8) and interactive exploration of data profiling results (Chapter 9) illustrate a common pattern of interactive applications tightly connecting them with the domain of adaptive physical database design: during the execution of queries we perform physical database design to support future interactions. In particular, for both crossfiltering and data profiling, we performed physical database design during queries that mainly involved group-by aggregations.

In this section, we extend our results beyond group-by aggregations by proposing instrumentation-based frameworks and techniques to allow applications to perform adaptive physical database design on selections and joins. For selections, we introduce instrumentation-based frameworks to ease the implementation of known and novel database cracking techniques without the need to alter the database internals (Section 10.1). For joins, we introduce instrumentation-based techniques for adaptive denormalization with the goal to optimize the performance of consecutive queries involving identical joins (Section 10.2).

## 10.1 Database Cracking

Database query engines evaluate range selection queries over base tables with serial scans, clustered or non-clustered indexed scans, or using binary search variants if the table is sorted on the attributes involved in the selection. Typically, serial selection scans are considered expensive when the selections have low selectivity, the predicate evaluation is expensive, or when scanned tuples are wide. In such cases, it is preferable to evaluate selections with either indexed scans or scans over sorted relations. The problem is that such indexes or sorted relations may not be available at the time of query execution, however. To address this problem, one approach could be to block the query execution to construct indexes or sort input tables. Unfortunately, such approaches are heavyweight and block the query execution for a lot of time. More importantly, sorting or providing a full index does not necessarily guarantee that queries in the workload will need the full power of an index or a sorted relation. For instance, in cases where users progressively zoom-in into a specific range, by tightening the bounds of the range selection query, only a small subset of the initial table will be used. Hence, time spent on full indexing or sorting is wasted.

For these reasons, database cracking [SDL18; SJD13; PPI+14; HIKY12; IKM07a; IKM07b; IMKG11; KM05; PIM15] has been introduced as a partial indexing and sorting paradigm that enables fast access to data partitions relevant to future range selection queries by adapting to the user workload. To illustrate, consider the following example:

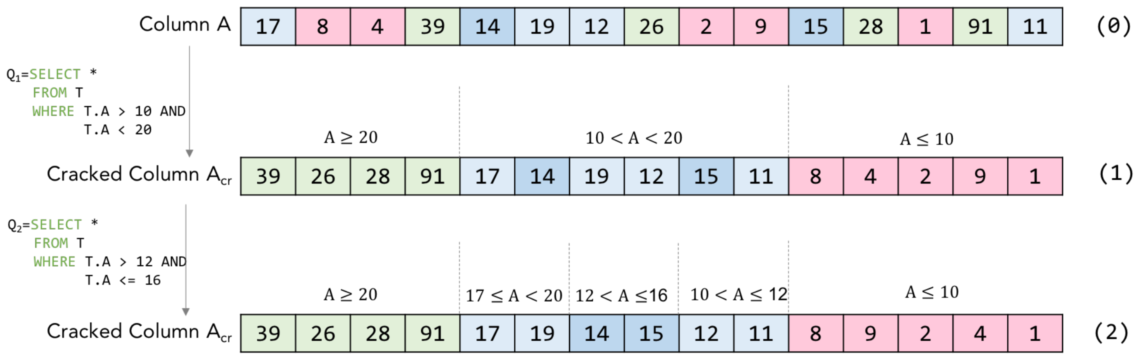


Figure 10.1: Cracking example.

**Example 14 (Cracking Example)** Consider the column A in relation T in Figure 10.1. An initial query $Q_1$ asks for the tuples in T where $A \in (10, 20)$. A database cracking technique will answer the query and at the same time reorganize the column A so that all values above, below, and within this range will be adjacent as shown in Figure 10.1(1). For this first query, the values will be stored in a separate column $\mathrm{A_{cr}}$, often called the cracked column. Next, a query $\mathrm{Q_2}$ that asks for the records in T in the range $\mathrm{A} \in (7, 16]$ can focus only on the last two partitions since the first partition is guaranteed not to have records in this range. Recursively, the database cracking technique can split the partitions involved in the current query as shown in Figure 10.1(2).

An important observation from the example above is that the query $\mathrm{Q_2}$ attempts to zoom into the results of $\mathrm{Q_1}$ by tightening its bounds (i.e., from $(10, 20)$ to $(12, 16]$). Similarly, one could also imagine several other access patterns (e.g., zooming-out or sequential access patterns). This illustrates a desired characteristic from database cracking techniques, namely, their robustness to different workloads. To this end, several database cracking techniques [HIKY12; SDL18; PIM15] have been introduced each demonstrating various robustness characteristics for different access patterns. Fitting arbitrarily complex access patterns, however, and designing novel database cracking techniques is an ever-increasing challenging task both in terms of algorithmic development and co-designing with database internals because database cracking is a by-product of query execution.

To this end, in this section, we focus on the problem of how can we enable the development of novel adaptive physical database designers for database cracking purposes. More precisely, we break down the problems involved in the construction of a database cracking technique. Then, we introduce two general instrumentation-based frameworks for the integration of cracking techniques within selection plans. Throughout, we discuss how known cracking techniques can be introduced in the instrumentation-based frameworks while our experiments show how instrumentation can introduce novel functionality that cracking techniques can use for optimization purposes. Our main result, that highlights the power of instrumentation, is that advanced database cracking techniques can be seamlessly integrated and developed within a database engine without changing the database internals.

### 10.1.1 Database Cracking Breakdown

At their core, database cracking techniques [SDL18; SJD13; PPI$^+$14; HIKY12; IKM07a; IKM07b; IMKG11; KM05] reorganize columns, and their corresponding tables, involved in selection queries to optimize the performance of subsequent selection queries. Hence, a natural way to breakdown how database cracking techniques operate is by understanding *how* they reorganize columns and tables involved in selections. Furthermore, since such techniques are triggered from the execution of selection queries, another way to break them down is by understanding *when* they perform the reorganization by means of what they perform *before*, *during*, and *after* the current selection query.

Next, we provide background and breakdown cracking techniques based on how they reorganize the physical database design. Then, we discuss what they perform before, during, and after the current selection. After this discussion, we summarize the requirements of database cracking techniques from instrumentation frameworks. The next section will introduce such instrumentation frameworks and show how we can fulfill these requirements.

#### Cracking-Driven Reorganization

Database cracking techniques reorganize columns involved in selections by solving variants of the well-known Dutch National Flag (DNF), or three-way partitioning, problem introduced by Edgar Dijkstra [Dij97, Chapter 14] that we rephrase as follows:

---

**Definition 2 (Dutch National Flag Problem (DNF))** Given an array of N elements with values {red, blue, green} randomly shuffled devise an efficient algorithm that sorts the array so that elements of the same color are all adjacent.

---

**Reduction from DNF.** To see the reduction from DNF to database cracking, consider each element in an array and a range query. Each element in the array is either above (red), within (blue), or below (green) the range. Database cracking, similarly to the DNF problem, asks to group together the elements that are above, within, and below a range.

**Crack-in-two and crack-in-three.** The 3-way partitioning of DNF can either be solved by algorithms that perform either directly 3-way partitioning or by two successive 2-way partitioning steps. In database cracking terminology, three-way partitioning algorithms are referred to as crack-in-three, and the ones that apply two consecutive 2-way partitioning steps are referred to as crack-in-two. For instance, if a range query [low, high] is given, then one can perform the 3-way partitioning by first performing a 2-way partitioning based on low and then another 2-way partitioning based on high.

**Partitioning Variants.** Consider again our zoom-in example in Figure 10.1. If zoom-in is the pattern of choice for the user exploration, it is clear that a lot of time is wasted by crack-in-two and crack-in-three algorithms in generating the partitions above and below the given range because subsequent queries will not access these partitions. This illustrates the major problem of initial database cracking techniques that decided the partitioning based on the selection predicate of the current query. This decision should be a function of both the current query and the expected future query workloads. To this end, several cracking techniques (e,g., hybrid cracking [IMKG11], stochastic cracking [HIKY12], and meta-adaptive indexing [SDL18]) have been introduced with the goal set to decide on partitioning strategies targeting various future workloads. Since future workloads may be arbitrarily complex, however, we expect novel database cracking techniques to condition further their partitioning strategies based on future workloads. In this direction, our main focus is to show how instrumentation can assist in the introduction of arbitrarily complex cracking techniques within a database in a principled manner.

**Out-of-place and in-place.** Another major concern with database cracking regards whether the reorganization is out-of-place (i.e., the partitions are stored in places other than the column from which they originated) or in-place (i.e., the partitions are stored in the same column from which they originated). Typically, the first query that triggers the column reorganization results in the generation of a new column, namely, cracked column, where the partitions will be stored. Hence, in this case, the reorganization is out-of-place. For subsequent selections, database cracking techniques typically perform in-place cracking on the new cracked column. However, we expect techniques to perform out-of-place cracking on cracked columns as well. This is because in-place cracking is typically more expensive than out-of-place as the latter can use more space to avoid several culprits of the former (e.g., immense data shuffling due to swaps).

**Tuple reconstruction and table reorganization.** Consider the queries of our example in Figure 10.1. Both queries perform a range selection on column A. However, the projection is on every column that appears under table A. This means that subsequent selections on column A will use the reorganized column A to perform the selection but access to the other columns is still required to provide the results. This can be addressed by maintaining an extra array with rids, to map values of the cracked column A to their original positions [SDL18], or techniques such as sideways cracking [IKM09] that physically reorganize copies of other columns so that they can be in sync with the cracked column. The former techniques can use the rid array to perform secondary index scans to base records to evaluate the projection. The latter techniques perform scans and evaluate projections on the reorganized columns. While these techniques are important for the optimization of future queries, they are agnostic to the future workload and assume no idle time in-between selection queries to better decide on the reorganization of the physical database design.

## Cracking Before, During, and After Query Execution

Besides how cracking techniques reorganize the physical design of columns and tables, another dimension to classify them is based on what they perform before, during, and after the execution of a selection query.

**Before.** Before the evaluation of a selection query, cracking techniques may need to perform monitoring of input queries to the database system, block the query execution to perform physical database design that was scheduled after previous selection queries, or even perform reorganization right before the query execution.

**During.** During the execution of selection queries, database cracking techniques typically either overlap the reorganization of columns with the query execution or compute statistics to be used after the query execution and before the next selection to perform the reorganization.

**After.** Finally, cracking techniques that perform cracking after query execution typically use statistics collected during the execution of the selection queries to better drive how they perform the reorganization.

### Cracking Requirements

Our discussion above illustrates that cracking techniques can be arbitrarily complex and database engines need to provide extensible and flexible ways for their principled introduction. More specifically, our discussion over the cracking reorganization illustrates that cracking techniques can be arbitrarily complex by targeting different database settings and future query workloads. Hence, both current and potentially future cracking techniques need flexibility for their introduction within the query execution. Furthermore, our discussion over when cracking takes places illustrates that cracking techniques need flexibility for when to focus their reorganization efforts.

Per current practice, cracking techniques have been predominantly introduced within MonetDB or as standalone tools, with the majority of other database engines to lack such functionalities. Cracking techniques, however, are increasingly important especially with respect to the advent of data-intensive interactive applications that need databases to reorganize the underlying storage adaptively and streamline future data-intensive workloads. As user exploration patterns can be arbitrarily complex, cracking techniques that aim to be robust towards such patterns can also become arbitrarily complex (e.g., instead of cracking a whole column we may want to crack only some subsets that best fit future user exploration patterns). As such, we expect the introduction of both known and novel cracking techniques within other databases aiming to target such application domains. In this direction, we next introduce instrumentation-based frameworks to show principled and flexible ways for the implementation of cracking techniques within an instrumentation-enabled database engine.

### 10.1.2 Instrumentation-Based Cracking Frameworks

In this section, we describe how database cracking can be implemented within SMOKE using our instrumentation framework. More specifically, our main focus is to introduce two cracking frameworks that inject the reorganization during selections or defer it after the selections. After their introduction, we conclude this subsection with other instrumentation mechanisms that cracking techniques may seek to use for their implementation.

#### Injected Cracking

Consider the instrumentation of the selection operator in Section 6.3.1. Our framework introduced three instrumentation points (i.e., $\sigma_{\mathrm{P}}^{\mathtt{before}}$, $\sigma_{\mathrm{P}}^{\mathtt{after}}$, and $\sigma_{\mathrm{N}}$) so that instrumentors can consume the tuples that satisfied and did not satisfy the predicate. Our goal here is to show that database cracking techniques can use these instrumentation points to perform their cracking logic. Consider the following code sketch and associated plans for the range selection query and the cracking instrumentor.

```
Column Tc_in, Tc_above, Tc_below, Tc;
for(Tuple t in T){
  if(t.a > 10 && t.a < 20){
    projection.consume(t)
    Tc_in.push_back(t.a)
  }else{
    if(t.a>=20)
      Tc_above.push_back(t.a)
    else
      Tc_below.push_back(t.a)
  }}
Tc = union(Tc_below, Tc_in, Tc_above)
```

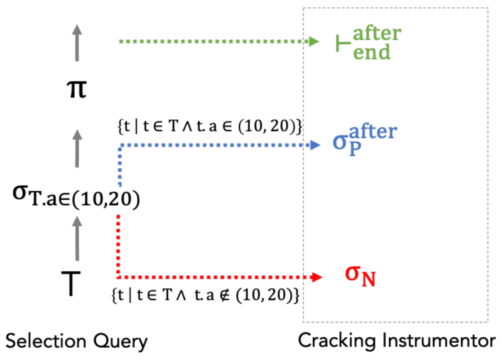


Figure 10.2: Code fragments in blue, green, and red denote code injected by the cracking instrumentor.

The query asks for the records in table $\mathrm{T}$ s.t. $\mathrm{T.a} \in (10, 20)$. The database compiles this query and generates a physical plan, as shown in the middle of Figure 10.2. The cracking instrumentor implements the $\sigma_{\mathrm{P}}^{\mathtt{after}}$, $\sigma_{\mathrm{N}}$, and $\vdash_{\mathtt{end}}^{\mathtt{after}}$ instrumentation points. $\sigma_{\mathrm{P}}^{\mathtt{after}}$

defines a column $\mathrm{Tc\_in}$ and populates it with the values of $\mathrm{T.a}$ that satisfy the predicate. $\sigma_{\mathrm{N}}$ defines two other columns Tc_above and Tc_below and populates them with the values of $\mathrm{T.a}$ that are above and below the range, respectively. These three columns essentially provide the three-way partitioning required by the database cracking. To conclude, the cracking instrumentor instruments the pipeline of the selection scan by implementing the $\vdash^{\mathtt{after}}_{\mathtt{end}}$ instrumentation point, that we introduced in Section 6.3. This operator unions the three columns in order to provide the cracked column Tc and is executed after the pipeline that involves the selection has finished its execution.

**Properties.** The cracking framework described above has many desirable properties:

1. The cracking implementation is up to the cracking instrumentor and it does not require cracking developers to change any of the database internals on their own. This highlights the overall power of instrumentation-enabled engines.

2. The framework provides flexibility for the introduction of both well-known and novel semantics. For instance, one could change the instrumentation logic to generate tables with all or some of the attributes of the input table. In database cracking terms, this corresponds to sideways projections that aim to avoid the tuple reconstruction costs due to non-clustered nature of the cracked column, as we discussed in Section 10.1.1. Furthermore, a cracking technique may decide to only materialize the partition of records that satisfy the selection (e.g., by instrumenting only $\sigma_{\mathrm{P}}^{\mathtt{after}}$ without instrumenting $\sigma_{\mathrm{N}}$) or even materialize only a subset of the partition of records that satisfy the selection (e.g., by implementing $\sigma_{\mathrm{P}}^{\mathtt{after}}$ to first check if a record belongs in the desired subset).

3. The framework also provides flexibility on scheduling the partitioning strategy. For instance, one could devise multiple variants by deferring the instrumentation of either the positive or the negative side. Deferring in this setup means that the selection will be re-executed after the execution of the initial selection but this time only for cracking purposes. Hence, one could inject cracking on the positive side and defer the

negative one and vice versa, or even inject the cracking for the negative side only for the values above the range and defer the ones below the range.

4. The final union to consolidate a cracking column happens after the selection. In fact, if the selection is part of a larger plan, this consolidation could happen after the whole plan, right after the pipeline that the selection is involved, or right after (or before) the execution of any pipeline (that follows the pipeline involving the selection) in the plan. The decision for when to consolidate is up to the cracking instrumentor.

5. The final consolidation could also not happen at all. The three partitions could be registered as individuals partitions of the cracked column. Future selection queries could simply scan these individual partitions without requiring access to specific parts of the cracked column.

6. The selection preserves the structure of the initial query. This means that parent operators can consume records that satisfy the selection without blocking on in-place cracking variants to finish their reorganization.

7. The actual partitioning is out-of-place which, although requires more memory, avoids swaps that hurt the performance of in-place cracking variants.

## Injected Statistics - Deferred Cracking

The scheduling flexibility discussed above (Property 3) is an important property because injecting cracking adds overhead to the query execution that upsteam applications may not tolerate. In this direction, deferring the cracking logic comes with several benefits for optimizations and policy-making that could be informed by lightweight statistics injected in the initial selection. This is the main idea behind the second framework for cracking purposes that we introduce here.

Consider the example code sketch for a cracking technique illustrated in Figure 10.3. Using the instrumentation points on selection, we can implement $\sigma_{\mathrm{P}}^{\texttt{after}}$ and $\sigma_{\mathrm{N}}$ to get

```
int Tc_in_cnt=0, Tc_below_cnt=0;
for(Tuple t in T){
  if(t.a > 10 && t.a < 20){
    projection.consume(t)
    Tc_in_cnt++
  }else{
    if(t.a<=10) Tc_below_cnt++
  }}
Column Tc = Column(T.size())
int Tc_below_idx = 0,
   Tc_in_idx = Tc_below_cnt,
   Tc_above_idx = Tc_below_cnt +
                 Tc_in_cnt;
for(Tuple t in T){
  if(t.a > 10 && t.a < 20)
    Tc[Tc_in_idx++] = T.a;
  else if(t.a >= 20)
    Tc[Tc_above_idx++] = T.a;
  else
    Tc[Tc_below_idx++] = T.a;}
```

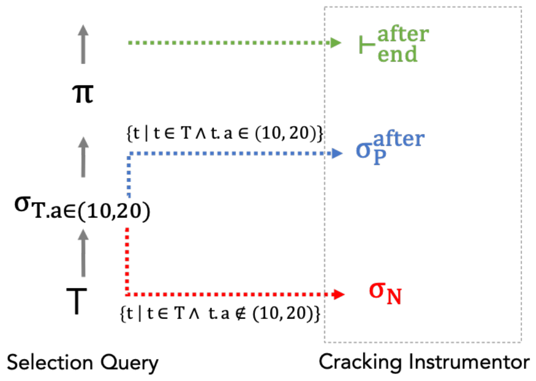

$\vdash^{after}_{end}$
$\pi$
$\{t \mid t \in T \wedge t.a \in (10,20)\}$
$\sigma_P^{after}$
$\sigma_{T.a \in (10,20)}$
$\sigma_N$
T
$\{t \mid t \in T \wedge t.a \notin (10,20)\}$
Selection Query
Cracking Instrumentor

Figure 10.3: Example of deferred cracking driven by statistics injected in the selection.

statistics from the query execution. In our example in Figure 10.3, we are getting the number of records that satisfy the selection (by implementing $\sigma_P^{\mathtt{after}}$ accordingly) and the number of records that are below the given range (by implementing $\sigma_N$ accordingly). Then, the cracking instrumentor can instrument the pipeline and defer the cracking reorganization after the selection. After the selection, the cracking instrumentor knows how many records are above, in, and below the range. Using this information, the choice of the reorganization can be better informed. For instance, using these statistics we can preallocate exact memory for the partitions, as shown in Figure 10.3. Similarly, we could decide on only cracking one of the partitions because the selectivities for the others are low, and so on.

The deferred reorganization can also be further informed by statistics gained at the coarse grain. For instance, if a sequential access pattern is the choice of the user for exploration, then instead of getting statistics for the partitions based on the current selection, a cracking technique could decide to implement $\sigma_P^{\mathtt{after}}$ and $\sigma_N$ to get statistics for the partitions that will be accessed next. Lets say that at the coarse level we have determined that the user is performing window-based selections with a step of 5. The current selection is on $\mathrm{T.a} \in (10, 20)$. This means that we should implement $\sigma_{\mathrm{P}}^{\texttt{after}}$ as $\mathrm{if(t.a > 15)Tc\_in\_cnt++;}$. Similarly, we can implement $\sigma_{\mathrm{N}}$ as $\mathrm{if(t.a < 25)Tc\_above\_cnt++;}$. Then, the deferred logic knows the size of the partition that will be accessed next and properly create a partition for this chunk. To conclude our discussion on the second framework, we also note that the properties that we introduced for the first framework are also provided by this framework.

### Other Useful Instrumentation Mechanisms

We conclude our discussion on cracking, with several other instrumentation mechanisms that are useful for the introduction of cracking techniques within a database. Throughout our discussion note that no change of database internals is required: the cracking module registers its logic to the instrumentation framework without worrying about the internals of the database engine—which is the overall point of our discussion. Furthermore, note that for our discussion next, we do not limit to instrumentation of selections in physical plans but also account for instrumentation at logical IR levels (i.e., SQL query, parsed SQL query, and logical plans) that we discussed in Chapter 2.

**Identifying selections.** When a query enters the database, the database cracking module should decide if the query is a selection query and if of interest for cracking purposes. The decision logic is up to the module and could be made either by instrumenting at the level of physical plans or at logical IR levels. For instance, the decision could be made by registering to the instrumentation points before or after the parsing module. Before the parsing module means that the cracking modules knows how to identify a selection in the textual representation of the SQL query while after the parsing module means that the cracking module could identify the selection in the parsed AST.

**Blocking the query execution.** Database cracking techniques should also be able to block the query execution in anticipation of the completion of a currently in-flight reorganization that could help either the selection query or further cracking decision. This blocking could happen before and after the parsing module, before or after the query optimizer, or right before the physical plan execution. Blocking before the optimizer is useful when the in-flight reorganization will affect the decisions of the optimizer while blocking after is typically due to a generated plan that is poor enough that waiting for the cracking reorganization pays off.

**Cracking before query execution.** Several cracking techniques perform a reorganization before the query execution. For instance, hybrid cracking could be triggered before the query execution to generate several partitions of a column. Similarly, a cracking technique may take it to the extreme and block the query execution to sort one or more columns directly. Such techniques may block the query execution and, upon completion, they could re-trigger the selection queries so that they can use the new physical database design. This is also allowed by blocking the query execution, stopping the current query execution, and spawning another query. All operations are supported by our instrumentation framework.

**Cracking with two-sided scans.** Our discussion on the instrumentation frameworks focuses on one-sided scans because this is the primary choice for selections by databases. Cracking techniques, however, may want to perform two-sided scans as they typically result in better performance on cracking reorganizations. If such scans are not available in a database, recall from our discussion in Chapter 6 that instrumentation applications can introduce their own physical operators. Furthermore, recall that our physical plan instrumentation framework provides essential operations for the replacement of operators within a physical plan. As such, cracking techniques can introduce two-sided scans, if not present in the database, and replace existing one-sided scans within physical plans. Finally, we note that the introduction of flexible defer and inject semantics on two-sided scans, similar to the ones that we introduced for one-sided scans, are interesting future work.

## 10.2 Denormalization

Many applications express their analytical logic using joins over normalized databases. Such joins can be very expensive—especially if they are repetitive and the applications have interactive latency requirements (e.g., interactive data visualizations). To account for the poor performance of joins, such applications first denormalize the database to generate a, so-called, denormalized representation using denormalization strategies [LP14; PDZ+18]. Then, joins over the normalized representation can be converted into scans over the denormalized one. Unfortunately, denormalized representations have two problems that we address in this section. First, inducing a denormalized representation is an expensive operation that may take a lot of time and space to construct. As such, it blocks the user exploration similarly to how data cubes, for crossfilter purposes, and sorting, for selection purposes, block the user exploration. Furthermore, denormalization comes with data redundancy costs and violates the physical data independence [Cod70], as we discussed in Section 7.5.4.

To account for these problems, we show the connection between provenance capture and denormalization and based on this connection we introduce a provenance-based denormalization technique. As we will see, fine-grained provenance capture on joins will provide us with a denormalized representation that is cheap to construct, avoids data redundancy costs, and does not require applications to change their querying logic (i.e., ensure physical data independence).

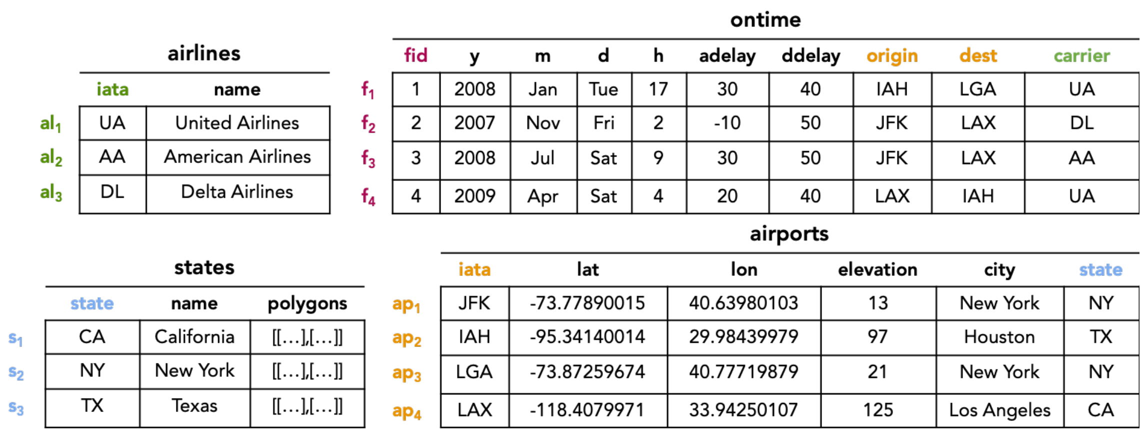
To ease our discussion over the different techniques, we consider a small instance of our example delayed flights database from Chapter 7. For convenience, Figure 10.4 repeats the schema from Chapter 7. Figure 10.5 depicts the database instance.

```
flights(fid,y,m,d,h,adelay,ddelay,origin,dest,carrier)
airlines(carrier,name,iata,active)
airports(apid,name,iata,lat,lon,elevation,city,state)
states(state,name,polygons[])
```

Figure 10.4: The flights database schema.

As an example denormalization of this database we will consider joining all the tables on the colored attributes:

Note that our goal here is not to introduce a denormalization strategy (i.e., what query to use to denormalize the database). Rather, our focus is given such a query how to create the



airlines

| | iata | name |
|---|---|---|
| $al_1$ | UA | United Airlines |
| $al_2$ | AA | American Airlines |
| $al_3$ | DL | Delta Airlines |

ontime

| | fid | y | m | d | h | adelay | ddelay | origin | dest | carrier |
|---|---|---|---|---|---|---|---|---|---|---|
| $f_1$ | 1 | 2008 | Jan | Tue | 17 | 30 | 40 | IAH | LGA | UA |
| $f_2$ | 2 | 2007 | Nov | Fri | 2 | -10 | 50 | JFK | LAX | DL |
| $f_3$ | 3 | 2008 | Jul | Sat | 9 | 30 | 50 | JFK | LAX | AA |
| $f_4$ | 4 | 2009 | Apr | Sat | 4 | 20 | 40 | LAX | IAH | UA |

states

| | state | name | polygons |
|---|---|---|---|
| $s_1$ | CA | California | [[…],[…]] |
| $s_2$ | NY | New York | [[…],[…]] |
| $s_3$ | TX | Texas | [[…],[…]] |

airports

| | iata | lat | lon | elevation | city | state |
|---|---|---|---|---|---|---|
| $ap_1$ | JFK | -73.77890015 | 40.63980103 | 13 | New York | NY |
| $ap_2$ | IAH | -95.34140014 | 29.98439979 | 97 | Houston | TX |
| $ap_3$ | LGA | -73.87259674 | 40.77719879 | 21 | New York | NY |
| $ap_4$ | LAX | -118.4079971 | 33.94250107 | 125 | Los Angeles | CA |

Figure 10.5: A small instance of the flights database schema.

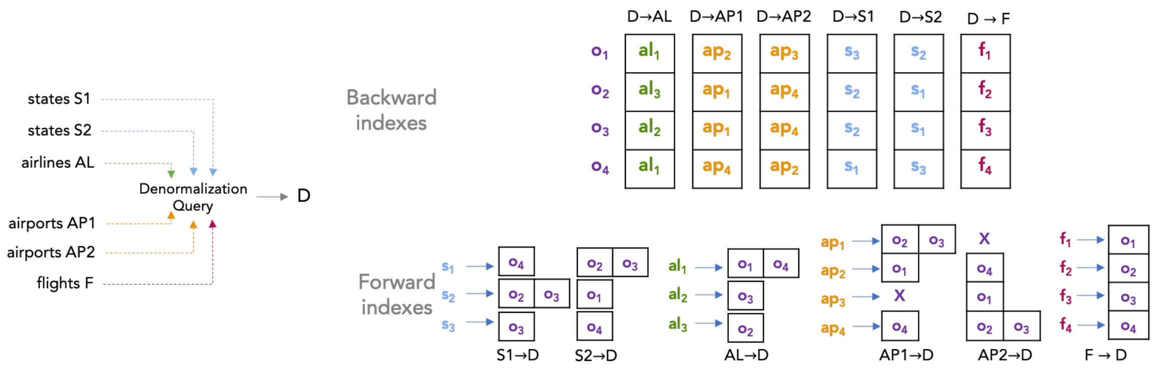
```
D = SELECT *
    FROM   flights F, airports AP1, airports AP2,
           states S1, states S2, airlines AL
    WHERE  F.origin = AP1.origin AND
           F.dest = AP2.dest AND
           F.carrier = AL.carrier AND
           AP1.state = S1.state
           AP2.state = S2.state
```

Figure 10.6: Example denormalization query for the flights database.

denormalized representation in such a way that the construction cost is low and the query performance over the denormalized relation is close or better that materializing directly the whole denormalized relation. Hence we note that denormalization queries that involve different types of joins (e.g., outer joins such as the ones introduced by WIDETABLE), projection clauses other than ‘*’ even if these involve functions over base attributes (e.g., a binning function over the delay attribute of the flights table), or even selections on base tables or intermediate join results are valid input for the techniques that we discuss below.

### 10.2.1 Provenance for Denormalization

Our core idea is that fine-grained provenance indexes constructed as a result of the denormalization query provide a tuple graph connecting the tuples of the output join result with the tuples of the base tables involved in the join computation. More precisely, the backward indexes provide a mapping from each output tuple to the input tuples that contributed to the output tuple, while forward indexes provide the inverse mapping (i.e., mapping from each input tuple to the output tuples that it contributed to). Hence, instead of materializing the denormalized representation with all the attributes from each table which has substantial materialization cost, we instead only materialize the forward and backward indexes.



(a) Denormalization Query

(b) Forward and backward indexes generated for the denormalized query

Figure 10.7: (a) The example denormalization query illustrated as a workflow. (b) Denormalized representation generated as a result of provenance capture on the example denormalization query.

To illustrate, consider the denormalization query in Figure 10.6. The forward and backward indexes as a result of fine-grained provenance capture on this query are shown in Figure 10.7: The backward rid indexes D $\rightarrow$ AL, D $\rightarrow$ AP1, D $\rightarrow$ AP2, D $\rightarrow$ S, and D $\rightarrow$ F map the output rids of the denormalized relation D to the input rids of the table instances airlines AL, airports A1, airports A2, states S, and flights F, respectively. (Note that there is one backward rid index per input table instance and that each backward rid index is an rid array because the mapping from output to input for joins is 1 to 1). Similarly, the forward rid indexes S $\rightarrow$ D, AL $\rightarrow$ D, AP1 $\rightarrow$ D, AP2 $\rightarrow$ D, and F $\rightarrow$ D map the input rids of the input table instances to the output rids of the denormalized relation. (Again, note that there is one forward index per input table instance, and each forward index is an rid index because the mapping from input to output for joins is 1 to N.)

Now, note that the denormalization query asks for all the attributes of the input tables as a result of the `SELECT` clause. This leads to materializing a full-blown denormalized relation D that contains all the attributes of each table, which is what we want to avoid due to its high materialization cost. The idea here is that if we materialize the join indexes then the denormalized relation does not need to be materialized at all. This is because instead of accessing the attribute values as materialized in the denormalized relation D we can evaluate future queries on D by using only the backward and forward indexes with the techniques that we show next.

### 10.2.2 Querying

Our provenance-based technique above constructs a denormalized representation. How can applications use this denormalized representation to streamline future join queries, however?

In this direction, it is easy to see that our provenance indexes for joins result in the physical encoding of a fundamental data structure known as join indexes [Val87]. Assume two relations R and S. Tuples in R are uniquely identified by the surrogate key $\mathrm{r}$. Similarly, every records in S is uniquely identified by the surrogate key $\mathrm{s}$. Now, recall that a join index for a join between tables R and S is a relation, say, RS with tuples $(\mathrm{r}, \mathrm{s})$ based on the results of the join. In our case, the surrogate keys are rids. Furthermore, instead of physically representing join indexes as relations, we represent them using rid indexes and rid arrays.

Now, note that this physical representation of join indexes (i.e., using rid indexes and rid arrays) is well-known [WLPS17] and has been used for the construction of efficient join evaluation algorithms [WLPS17; LR99]. Hence, the main novelty of our technique is the construction of the denormalized representation during the execution of initial joins (as opposed to their so-far known offline construction) that well-known join algorithms can readily pickup. Furthermore, several other physical representations for join indexes have already been introduced, primarily involving compression [WLPS17], that stem from the same representation that our provenance-based technique induces. This highlights a potentially rich space for future work to address how to push the construction of alternative physical representations of join indexes (e.g., by pushing the compression algorithms in [WLPS17]) within the provenance capture phase.

## 10.3 Experimental Settings

Our experiments in this chapter seek to show the novel semantics and the performance benefits that instrumentation engines can provide to adaptive physical database design techniques. To this end, we show the performance of instrumentation-based techniques for database cracking and adaptive denormalization in both novel and standard settings.

**Datasets.** For our experiments, with both cracking and adaptive denormalization, we use common settings in terms of datasets. For cracking we compare techniques using a uniform dataset of 10 million double precision values, while for adaptive denormalization we compare techniques using TPC-H.

Table 10.1: Cracking techniques that we use in our evaluation.

| Abbreviation | Description |
|---|---|
| NO CRACKING | Query execution without cracking. |
| DEFER | Injected statistics, deferred cracking. |
| INJECT | Injected cracking. |
| INJECT-NEGATIVE | Injected cracking materializing only records not passing the selection. |
| INJECT-POSITIVE | Injected cracking materializing only records passing the selection. |
| INJECT-2SIDED | Cracking by instrumenting two-sided scans. |

**Compared Cracking Techniques.** Table 10.1 shows a brief description of the different techniques that we experiment with for cracking. NO CRACKING evaluates selection queries without performing cracking and forms our baseline. DEFER and INJECT refer to the techniques that we introduced in Section 10.1 for cracking injected within the selection and deferred after the selection, respectively. INJECT-POSITIVE and INJECT-NEGATIVE refer to techniques that perform cracking only for the records that pass or not the selection, respectively. Finally, INJECT-2SIDED refers to a standard cracking technique that uses two-sided scans (as opposed to the one-sided scans of INJECT). To introduce this technique in SMOKE we used our instrumentation framework to (1) introduce two-sided scans, (2) replace the one-sided scans provided by the selection operator, and (3) instrumented the two-sided scans to perform cracking. Steps (1) and (2) are implemented using the replace functionality of the Actions module and step (3) is implemented by instrumenting instrumentors that we briefly discussed in Section 6.4.4.

Table 10.2: Denormalization techniques that we use in our evaluation.

| **Abbreviation** | **Description** |
|---|---|
| NO MATERIALIZATION | No materialization of the join results. |
| FULLROW | Materializes the join results in row-store format. |
| FULLCOL | Materializes the join results in column-store format. |
| FULLROW+COL | Materializes the join results in row-store format followed by converting the row-store to column-store. |
| FULLROW+COMPESSION | Materializes the join results in row-store format followed by dictionary compression of textual attributes. |
| FULLCOL+COMPESSION | Materializes the join results in column-store format followed by dictionary compression of textual attributes. |
| BT+FT | Our provenance-based denormalization technique. |

**Compared Denormalization Techniques.** We compare the performance of denormalization techniques on denormalizing the join `Lineitem` $\bowtie$ `Orders` $\bowtie$ `Customer` $\bowtie$ `Nation` of the TPC-H database (SF=1). Table 10.2 show the techniques that we use for materializing denormalized representations. FULLROW materializes the result of the join in row-store format. FULLCOL materializes the result in columnar format. FULLROW+COL first materializes the result in row-store format and then converts it into column-store format. FULLROW+COMPESSION and FULLCOL+COMPESSION perform dictionary compression on the induced row- and column-store formats of FULLROW and FULLCOL. (We do not perform compression on the representation of FULLROW+COL: FULLROW+COL and FULLCOL result in the same representation, yet the former is slower to construct.) All the techniques we have described so far are the most common and performant denormalization techniques [PDZ+18; LP14]. (This is modulo techniques that perform further compression on other types of columns on either row- or column-store representations. We found that dictionary compression is already costly enough and we omit further compression steps.). Furthermore, BT+FT refers to our provenance-based technique from Section 10.2, that represents the denormalized relation using provenance indexes. Finally, NO MATERIAL-

IZATION performs the join but does not materializes any result and we use it only to report denormalization overheads. We have implemented and compared all techniques within SMOKE. Note that SMOKE is a row-store that does not support compression. We have extended SMOKE to support columnar storage and dictionary compression only to support the denormalization techniques that rely on this functionality.

**Measures.** We compare the different techniques on their absolute latency for completion of respective tasks as well as on the memory they use.

**Platform.** We ran our experiments on a MacBook Pro running macOS Sierra 10.14.1 with 16GB 2133MHz LPDDR3 memory (caches include 32KiB L1d, 32KiB L1i, 256KiB L2, and 4MiB L3) and a 2.3GHz Intel Core i5 processor.

## 10.4 Experimental Results

Having described our experimental settings, we next present our experimental results on database cracking (Section 10.4.1) and denormalization (Section 10.4.2)

### 10.4.1 Database Cracking

The goal of our experiments with database cracking is to highlight novel semantics that instrumentation provides for techniques aiming to address this problem. To this end, Figure 10.8 drives our discussion by showing the latency (y-axis) of cracking techniques while varying the selectivity (x-axis) of a range selection predicate ($\vartheta_1 < \mathrm{v} < \vartheta_2$). (The selection is applied over the 10 million double precision values of the dataset we described in our settings.) Our main observations are as follows:

Our first observation regards the comparison between INJECT, INJECT-POSITIVE, and INJECT-NEGATIVE. INJECT-POSITIVE and INJECT-NEGATIVE always outperform INJECT. This is because INJECT materializes the whole cracked column whereas INJECT-NEGATIVE materializes only the values that did not satisfy the selection and INJECT-POSITIVE materializes only the values that pass the selection. These results highlight that if we know

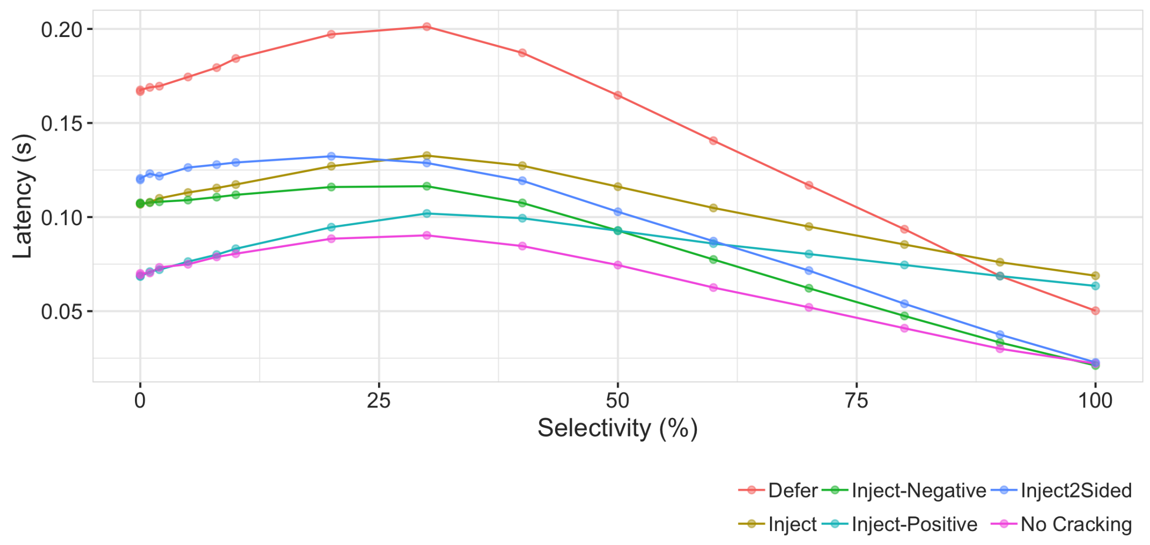


Figure 10.8: Latency of different cracking techniques.

exploration patterns that users follow, then we can use this knowledge to push (through instrumentation) more complicated cracking strategies in selections that a) best fit user-exploration patterns and b) decrease the overheads of materializing whole cracked columns that traditional cracking strategies perform.

Our second observation regards the comparison between INJECT-POSITIVE and INJECT-NEGATIVE. As shown in Figure 10.8, INJECT-POSITIVE is faster than INJECT-NEGATIVE for all selectivities below 50%. Above 50%, this observation is inverted and INJECT-NEGATIVE becomes faster than INJECT-POSITIVE. This is because INJECT-POSITIVE materializes the values that pass the predicate and INJECT-NEGATIVE materializes the values that did not pass the predicate. Hence, below 50% selectivity INJECT-POSITIVE has to perform less work than INJECT-NEGATIVE, above 50% the roles are inverted and INJECT-POSITIVE has to perform more work, and at 50% selectivity both approaches have the exact same performance, as shown in Figure 10.8.

Our third observation regards the bell curves that are formed by the different approaches while we vary the selectivity. This is due to branch mispredictions as is also noted in [PPI+14]. We note that to eliminate branch mispredictions a common technique is to use predication. Branch-free cracking variants that use predication to address this problem have already been introduced [PPI+14]. Such variants require significant rewriting of the underlying selections and are suitable for different types of workloads and selection selectivities. In this direction, we believe our physical plan instrumentation framework can help in the introduction of complex techniques (e.g., to introduce predication) by altering the internal logic of selections.

Our fourth observation regards the comparison between INJECT with DEFER. Note that the DEFER approach is slower than the INJECT approach for all selectivities below a threshold (i.e., ~85% selectivity in our experiments in Figure 10.8). Although DEFER is slower in most of the cases, recall that it is executed after the selection. Therefore, it does not block the query execution for cracking purposes as is the case for INJECT. Now the reason why DEFER is faster than INJECT is due to two reasons. First, recall that branch mispredictions are low for high selectivities. As a result, DEFER avoids both mispredictions in the initial selection and in the deferred execution. Second, recall that INJECT needs to reallocate memory during the initial selection to store the values that are above, below, and in the range of the selection. This is because the size of these arrays (i.e., below, above, and in the range) is not known in advance and INJECT needs to perform reallocations during the execution of the selection when appending to these arrays. In contrast, DEFER tracks the size of the arrays during selections. After the selection, the sizes of the arrays are known, and DEFER uses them to allocate the arrays with exact size. As branch mispredictions decrease, the benefit of allocating once using DEFER catches up the reallocations of INJECT, which overall renders DEFER better in high selectivities.

Our fifth observation regards the comparison between cracking using one-sided and two-sided scans (i.e., INJECT and INJECT-2SIDED). As shown in Figure 10.8, INJECT-2SIDED is worse than INJECT for small selectivities and better in high selectivities. This is due to the differences in branch mispredictions and cache misses of single- and two-sided scans as well as because INJECT-2SIDED performs in-place cracking and does not need to store an extra column. (Recall that INJECT is out-of-place and needs to store an extra column.) For our purposes, the main takeaway is that instrumentation can be equally used to express and introduce both variants in a principled way within a database without having to rewrite its internals—which is the overall point of our discussion.

So far, we have compared the different techniques with regards to their overall latency. For completeness and to conclude our discussion on cracking, we also compare techniques based on the space that they use. For our experiments, recall that the underlying column has N=10mil. double precision numbers. DEFER and INJECT needs to write a column the size of the original one (i.e., N). INJECT-NEGATIVE and INJECT-POSITIVE need to write $(1-\mathrm{X})*\mathrm{N}$ and $\mathrm{X}*\mathrm{N}$, respectively, where $\mathrm{X}$ refers to the selectivity. INJECT-2SIDED performs in-place cracking. Hence, it does not require extra storage. Finally, NO CRACKING performs the selection without materializing any information for database cracking purposes.

## 10.4.2 Denormalization

To evaluate our denormalization strategies, we experiment with TPC-H. More specifically, we compare our provenance-based denormalization technique (i.e., BT+FT) with the denormalization techniques FULLROW, FULLCOL, FULLROW+COL, FULLROW+COMPESSION, and FULLCOL+COMPESSION that we outlined in our settings (Section 10.3). Also, recall from our settings that the denormalization is on the join `Lineitem` ⋈ `Orders` ⋈ `Customer` ⋈ `Nation` over a TPC-H instance with SF=1. Figures 10.9 and 10.10 compare the techniques in terms of latency and space required to construct the denormalized representation. To better explain denormalization overheads, we also include the results of NO MATERIALIZATION that only performs the join without materializing a denormalized representation. Next, we discuss in detail our main observations over our experimental results.

**Latency of BT+FT.** Among the different techniques, BT+FT takes the least amount of time (i.e., 2.7s) to construct its denormalized representation. In comparison to NO MATERIALIZATION (i.e., not materializing a denormalized representation), BT+FT incurs only a ~1× overhead whereas the rest of the techniques (i.e., FULLROW, FULLCOL, FULLROW+COL, FULLCOL+COMPESSION, and FULLROW+COMPESSION) incur significantly

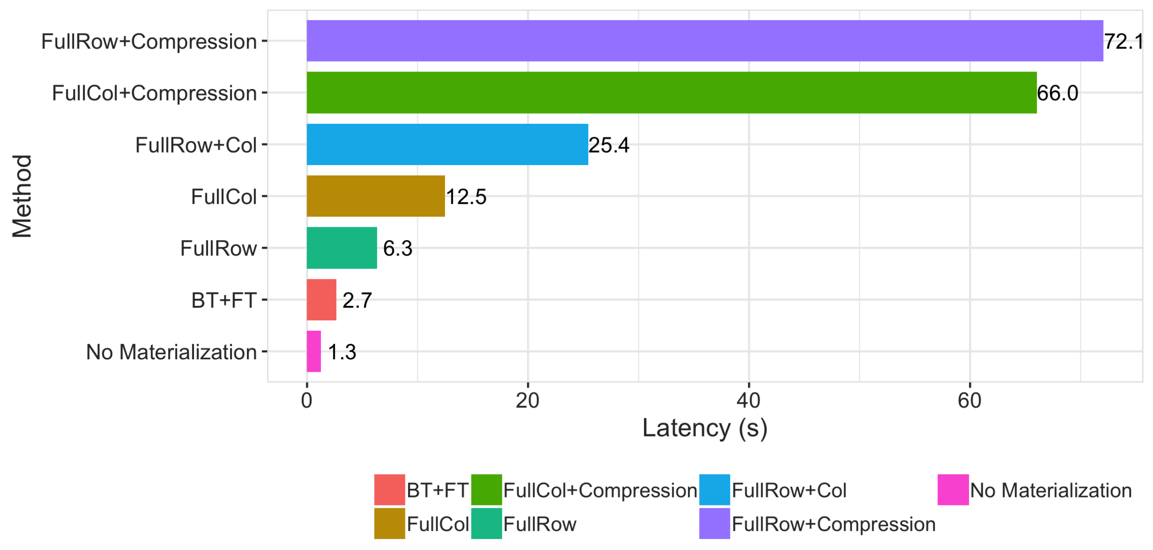


Figure 10.9: Latency of different denormalization techniques.

higher overheads (i.e., ~$3.8\times$, ~$8.6\times$, ~$18.5\times$, ~$49.7\times$, and ~$54.4\times$ overhead, respectively). These results highlight the latency-wise benefits of BT+FT for denormalization purposes.

**Comparison with FULLROW, FULLCOL, and FULLROW+COL.** The main reason why BT+FT outperforms FULLROW, FULLCOL, and FULLROW+COL is because the cost of constructing provenance indexes is significantly smaller than materializing full relations. In essence, the three techniques FULLROW, FULLCOL, and FULLROW+COL perform what we called LOGIC-TUP provenance capture in Section 3.7 which has significantly higher capture (and query) costs, as we showed experimentally in Section 3.7. Their main difference is on how the materialize their end-result (i.e., in column or row format) which is the main reason behind their overall differences on latency. Finally, note that BT+FT has the same complexity no matter how wide the denormalized relation becomes. In contrast, FULLROW, FULLCOL, and FULLROW+COL would have higher or lower overhead depending on whether the denormalized relation was wider of narrower, respectively. While the typical case is to have wide denormalized relations, we note that in order for FULLROW, FULLCOL, and FULLROW+COL to have the same or better performance than BT+FT, the

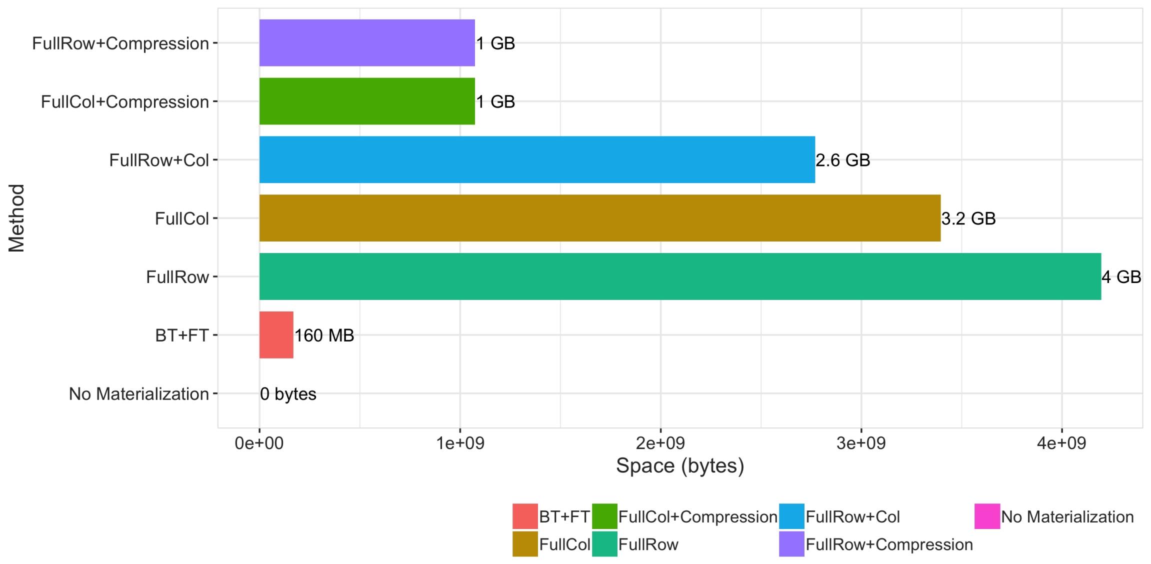


Figure 10.10: Space required by different denormalization techniques.

input relations to the join need to have records with size equal or less to the size of an RID. This is an extreme and highly unlikely case but we note it here for completeness.

**Comparison with FULLROW+COMPESSION and FULLCOL+COMPESSION.** FULLROW+COMPESSION and FULLCOL+COMPESSION perform dictionary compression on the textual attributes of the relations induced by FULLROW and FULLCOL, respectively. Since the dictionary compression is performed after the denormalization, FULLROW+COMPESSION and FULLCOL+COMPESSION incur even higher overhead than FULLROW and FULLCOL which, in turn, have higher overhead than BT+FT.

**Comparison on space consumption by different denormalization techniques.** Finally, we also compare the different denormalization techniques in terms of space consumption. As shown in Figure 10.10, BT+FT requires only 160MB for the provenance indexes while the compared techniques require at least 1GB (i.e., FULLROW+COMPESSION and FULLCOL+COMPESSION that perform dictionary compression; for these two techniques, we do not include the space required for the dictionary) and up to 4GB (i.e., FULLROW). These results highlight the space-wise benefits of BT+FT for denormalization purposes.

*Takeaways: Our experiments provide evidence that instrumentation-enabled database engines provide principle mechanisms for the introduction of both well-known as well as novel and performant database cracking and denormalization techniques.*

## 10.5 Conclusions

In this chapter, we explored connections between instrumentation (and provenance) with physical database techniques. Based on connections with database cracking, we introduced two novel instrumentation frameworks for database cracking purposes that can be used for the replication of well-known cracking techniques as well as for the introduction of novel cracking techniques. Furthermore, based on the connections between provenance and denormalization, we introduced a denormalization technique that constructs denormalized representations faster and using less space than well-known denormalization techniques. As such, our discussion and experiments highlight the expressive and optimization power of instrumentation-enabled engine in the domain of physical database design. Going forth, we believe that instrumentation can be central in the construction of novel physical database designs (e.g., online compression of provenance indexes by using [WLPS17] and devising list and bitmap compression algorithms to further optimize denormalization schemes) with the overall goal to best-fit user exploration patterns.

# Chapter 11

# Query Discovery

We continue our discussion on application domains by focusing on query discovery. Query discovery techniques aim to provide search interfaces on top of databases so that end users can discover queries that best fit their analytical needs. More specifically, given an input database and examples of desired query results through an interface, the goal is to return possible queries that generate the example results (or a superset of them). This formulation can be attractive because SQL queries are known to be hard to compose due to the compositionality of SQL. Hence, query discovery approaches typically focus on a semantically meaningful subset of SQL for which discovering queries can be efficient.

In this direction, this chapter presents a query discovery system, namely S4, that provides a spreadsheet-style search interface on top of analytical databases, so that end users can discover queries from an important class of join queries, namely, project-join queries.

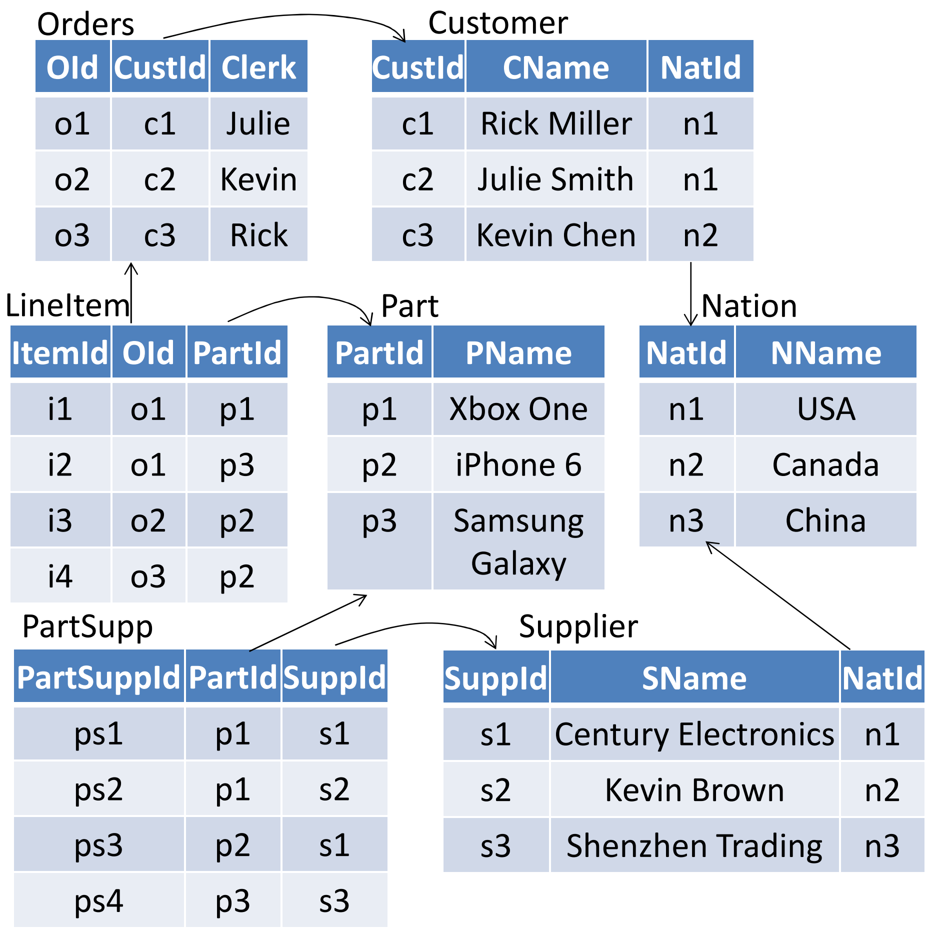
As a system, S4 precedes the development of SMOKE. As such, it provides us with an opportunity for a retrospective analysis of how we could have build S4 if instrumentation-enabled engines, such as SMOKE, were available. We present this analysis in Section 11.11 after the detailed presentation of S4. Finally, S4 is a product of my internship at Microsoft Research in collaboration and under the guidance of some of my great colleagues and mentors: Kaushik Chakrabarti, Surajit Chaudhuri, and Bolin Ding.

## 11.1 Introduction

Modern data warehouses usually have large and complex data schemas. A decision-support query on such a data warehouse typically touches a small portion of the schema. However, to express such a query, the enterprise information worker needs to comprehend the entire schema and locate elements of interest. This is extremely burdensome for most users.

Query discovery has recently been proposed as a solution to this problem [QCJ12; SCC$^+$14]. An enterprise information worker is often aware of *a few example tuples* that should be present in the output of a query. These example tuples together form an *example spreadsheet*, one per each row. Previous systems discover *project-join queries* (*PJ queries*) that contain the given example tuples, or the example spreadsheet, in their output [QCJ12; SCC$^+$14]. This liberates users from understanding the entire schema.

**Example 15 (Discovery of PJ queries in TPC-H)** Consider a database instance of a TPC-H sub-schema in Figure 11.1. The database contains information about customers, the countries they live in, the orders they placed, the parts purchased in each order, the suppliers of those parts, and the countries the suppliers are based in. The arrows point in the direction of foreign-key to primary-key relationships between pairs of relations. Suppose an enterprise information worker uses a query discovery system to discover the PJ query that outputs all customers and, for each customer, outputs her name, the name of the country she lives in, and the names of the parts she ordered. The PJ query and its output is shown in Figure 11.2(b)-(i). She is aware of an example spreadsheet of three example tuples that should be present in the query result: a customer named ‘Rick’ (does not know his full name) who lives in ’USA’ and ordered an ‘Xbox’, a customer named ‘Julie’ (not sure where she lives) who ordered an ‘iPhone’ and a customer named ‘Kevin’ who lives in ‘Canada’ (not sure what he ordered). She can provide this information by typing these example tuples into an example spreadsheet (e.g., in Microsoft Excel or Google Sheets) as shown in Figure 11.2(a). Note that some cells in the example spreadsheet can be empty. The system returns the desired PJ query in Figure 11.2(b)-(i) as it contains all the example tuples in its



Orders

| OId | CustId | Clerk |
|---|---|---|
| o1 | c1 | Julie |
| o2 | c2 | Kevin |
| o3 | c3 | Rick |

Customer

| CustId | CName | NatId |
|---|---|---|
| c1 | Rick Miller | n1 |
| c2 | Julie Smith | n1 |
| c3 | Kevin Chen | n2 |

LineItem

| ItemId | OId | PartId |
|---|---|---|
| i1 | o1 | p1 |
| i2 | o1 | p3 |
| i3 | o2 | p2 |
| i4 | o3 | p2 |

Part

| PartId | PName |
|---|---|
| p1 | Xbox One |
| p2 | iPhone 6 |
| p3 | Samsung Galaxy |

Nation

| NatId | NName |
|---|---|
| n1 | USA |
| n2 | Canada |
| n3 | China |

PartSupp

| PartSuppId | PartId | SuppId |
|---|---|---|
| ps1 | p1 | s1 |
| ps2 | p1 | s2 |
| ps3 | p2 | s1 |
| ps4 | p3 | s3 |

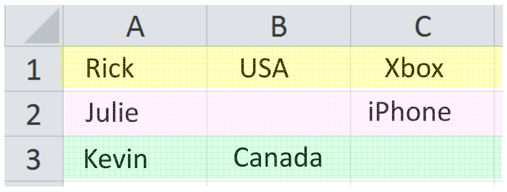
Supplier

| SuppId | SName | NatId |
|---|---|---|
| s1 | Century Electronics | n1 |
| s2 | Kevin Brown | n2 |
| s3 | Shenzhen Trading | n3 |

Figure 11.1: A sample database

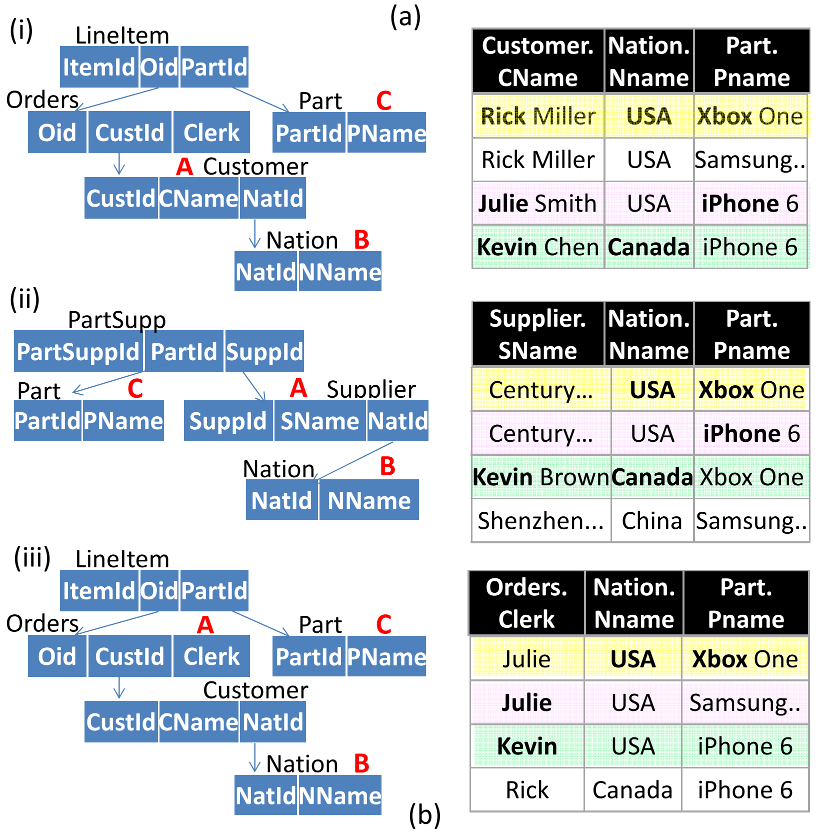
output relation. The example tuples and corresponding tuples in the output are shaded with the same color. The system maps the columns of the example spreadsheet to the projected columns in the query for users to better understand the PJ query discovered; the latter are labeled by the name of the corresponding column in the example spreadsheet (A, B and C).

One main limitation of these previous systems is that they require the output relations of PJ queries to *exactly* contain all the example tuples and do not perform any ranking. As a result, they cannot i) tolerate errors that the user might make while providing the example tuples and ii) perform IR-style relevance ranking.



|   | A | B | C |
|---|---|---|---|
| 1 | Rick | USA | Xbox |
| 2 | Julie |  | iPhone |
| 3 | Kevin | Canada |  |

(a)



| Customer. CName | Nation. Nname | Part. Pname |
|---|---|---|
| **Rick** Miller | **USA** | **Xbox** One |
| Rick Miller | USA | Samsung.. |
| **Julie** Smith | USA | **iPhone** 6 |
| **Kevin** Chen | **Canada** | iPhone 6 |

| Supplier. SName | Nation. Nname | Part. Pname |
|---|---|---|
| Century... | **USA** | **Xbox** One |
| Century... | USA | **iPhone** 6 |
| **Kevin** Brown | **Canada** | Xbox One |
| Shenzhen... | China | Samsung.. |

| Orders. Clerk | Nation. Nname | Part. Pname |
|---|---|---|
| Julie | **USA** | **Xbox** One |
| **Julie** | USA | Samsung.. |
| **Kevin** | USA | iPhone 6 |
| Rick | Canada | iPhone 6 |

Figure 11.2: (a) Example spreadsheet. (b) PJ queries and their outputs.

- *Tolerating errors*: Suppose the user wants to discover the query that outputs all orders and, for each order, outputs the name of clerk who processed the order, the country of the customer who placed the order, and the parts in the order. She provides an example tuple: a clerk named 'Rick' processed an order from a customer in 'USA', and the order consisted of the part 'Xbox'. However, it is not 'Rick' but another clerk 'Julie' who processed that order. The desired PJ query and its output is shown in Figure 11.2(b)-(iii). We say that she made a *relationship error* with respect to that PJ query as, although 'Rick' is a correct domain value (he is indeed a clerk), the provided relationship of 'Rick' with 'USA' and ‘Xbox’ is wrong. The user can also make *domain errors*. Suppose the user wants to discover the query that outputs all suppliers and, for each supplier, outputs its name, the country it is based in and the parts it supplies. She then provides such an example tuple: a supplier named ‘Rick’ based in ‘USA’ who supplies the part ‘Xbox’. The desired PJ query and its output is shown in Figure 11.2(b)-(ii). There is a domain error with respect to the PJ query as there is no supplier named ‘Rick’.

- *Performing relevance ranking*: Suppose there is a supplier with name ‘Welton USA’ who supplies Xbox One. Consider the first example tuple in Figure 11.2(a). In addition to the the PJ query shown in Figure 11.2(b)-(i), some other PJ queries may also contain that example tuple in their outputs (e.g., a customer ‘Rick Miller’ who ordered ‘Xbox One’ which is supplied by ‘Welton USA’). The former is more relevant, as country name ‘USA’ is a better match to ‘USA’ in the example tuple than supplier name ‘Welton USA’.

To address the above issues, in this chapter, we propose to *discover not only the PJ queries which exactly contain the given example tuples, or the example spreadsheet, in its output but also those that partially contain them.* We compute a relevance score for each PJ query that quantifies how well its output contains the example tuples and return *PJ queries with the top-*k *highest scores*.

## Technical challenges

In a large real-world database, there are numerous ways of projecting and connecting tables and rows through foreign keys. So there could be millions of PJ queries that partially contain the user-specified example tuples in their output relations.

The first technical challenge is to develop a scoring model that allows us to tolerate relationship/domain errors and quantifies how well the user-given example spreadsheet is contained in the output of PJ queries, in order to perform a relevance ranking of them.

The second and main technical challenge is to compute the top-k PJ queries *efficiently*. One important application of our system is to provide *online* data-search and discovery services in data processing tools such Excel Online [Exc18] and Google Sheets [Goo18]. While a user may spend a significant amount of time in specifying the query, such as the example spreadsheet in our system, it has been shown that, in the context of online search, query latency is critical to user satisfaction. Increases in latency directly lead to lower utilization and higher rates of query abandonment [Bru09; May06].

## Overview of our solution and key insights

In this chapter, we adapt a *candidate-enumeration and evaluation* framework, which is also used in keyword search systems for relational databases [ACD02; HGP03; LLWZ07].

In the first step, called *PJ query enumeration*, we enumerate all candidate PJ queries that are potential answers for a user-specified *example spreadsheet*. The only requirement for these candidate PJ queries, called *minimality*, is that no table or projection column can be dropped without "losing" in relevance score. In the second step, called *PJ query evaluation*, we execute candidate PJ queries, and compare their output relations with the example spreadsheet to calculate their scores. A naive solution is to execute all the candidates and output the top-$k$ with the highest scores.

It is important to note that the first step is very efficient, as it is pursued on the schema-level and no join is required. So it constitutes a negligible fraction of the overall query processing time. The second step, evaluating scores of PJ queries, is expensive (as it requires joins). As our scoring model quantifies how well the user-given example spreadsheet is contained in the output of join and projection, we need to at least examine rows in the join output, which may partially contain an example tuple. So this challenge translates to that of *evaluating as few PJ queries as possible*. The naive solution, evaluating all the candidates, is hence infeasible (we will compare it with the approaches we propose in Section 11.6).

Although calculating the exact relevance scores is expensive, we derive their upper bounds in a much more efficient way (without executing any join). Inspired by the work on multi-step kNN search [SK98], we evaluate PJ queries in decreasing order of their upper bound scores, and terminate with the top-$k$ as soon as the max upper-bound score of

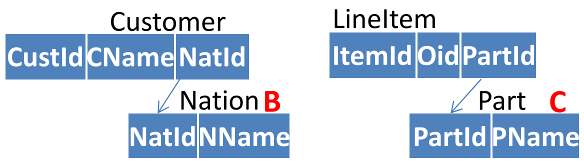


Figure 11.3: Common sub-expressions (sub-PJ queries) in Figure 11.2(b)

non-evaluated queries is no higher than the current top-k score. We refer to the resulting approach as BASELINE.

Our main insight to improve on BASELINE is that there are many common sub-expressions, called *sub-PJ queries*, that are *shared* among the PJ queries. For example, the PJ queries (i) and (iii) in 11.2(b) share the two sub-PJ queries shown in Figure 11.3. If we can compute the output relations of these sub-PJ queries once, and cache (or memorize) them in memory, we can re-use them multiple times later when we evaluate queries containing these two sub-PJ queries. This reduces the overall evaluation cost significantly.

A novel component, called *caching-evaluation scheduler*, in our system determines, for the set of candidate PJ queries, i) the order following which these PJ queries will be evaluated; ii) output relations of which sub-PJ queries to be cached; and iii) when to put the output relations into the cache and when to remove them, as we have only a budgeted amount of memory. There are two aspects in the objective of this component: one is to evaluate *as few PJ queries as possible* to discover the top-k; and the other one is to utilize the cached output relations as much as possible to reduce the overall *evaluation cost*. We will formalize the task of this component and refer to it as *caching-evaluation scheduling problem*.

**Contributions and organization.** We have built a *spreadsheet-style search system*, namely, S4, to tackle the challenges based on the above insights. Our contributions are as follows:

- We introduce a novel scoring model for a PJ query w.r.t. an example spreadsheet. It allows us to tolerate both types of errors and perform IR-style relevance ranking of PJ queries in response to example spreadsheets. (Section 11.2)

- We introduce S4 based on a candidate-enumeration and evaluation framework, and we enable a flexible caching-evaluation component in its architecture. (Section 11.3)
- To tackle the technical challenges of our task, we first introduce some basic operators in our system and propose our BASELINE strategy with the goal set to evaluate as few PJ queries as possible. (Section 11.4)
- We then introduce our cache-aware optimization techniques to improve on the BASELINE. We propose the *caching-evaluation scheduling problem* with the objective of minimizing the overall evaluation cost. We prove it is NP-complete. Several novel heuristics are proposed to solve this problem, and the resulting strategy is called FASTTOPK. We prove that FASTTOPK has performance guarantee in the worst-case in two aspects: i) it does not evaluate too many PJ queries in addition to the necessary ones; and ii) the gap between the evaluation cost introduced by FASTTOPK strategy and the optimal evaluation cost is bounded in the worst case. We also introduce how to extend our system and strategies to handle incremental updates on the example spreadsheet. (Section 11.5)
- We perform our experimental study on both real-life and synthetic datasets to evaluate the efficiency of our approaches, together with a user study to evaluate the effectiveness of our scoring model. (Section 11.6)

Finally, extensions and proofs are presented in Sections 11.7 to 11.9.

## 11.2 System Task and Scoring Model

We first present our data model and formally define our system task of *discovering top-k project-join queries for a given example spreadsheet*. Then, we present the model to compute the relevance score of a project-join query w.r.t. an example spreadsheet.

### 11.2.1 Data Model

We consider a database $\mathcal{D}$ with $\mathrm{m}$ relations $\mathrm{R}_1, \mathrm{R}_2, \cdots, \mathrm{R}_\mathrm{m}$. For a relation $\mathrm{R}$, let $\mathrm{R}[\mathrm{i}]$ denote its $\mathrm{i}^{\mathrm{th}}$ column, and $\mathrm{col}(\mathrm{R}) = \{\mathrm{R}[\mathrm{i}]\}_{\mathrm{i}=1,\ldots|\mathrm{col}(\mathrm{R})|}$ denote the set of columns of $\mathrm{R}$. For a tuple $\mathrm{r}$ in $\mathrm{R}$, denote $\mathrm{r} \in \mathrm{R}$ and let $\mathrm{r}[\mathrm{i}]$ be its *cell* value on the column $\mathrm{R}[\mathrm{i}]$.

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ denote the *directed schema graph* of $\mathcal{D}$ where the vertices in $\mathcal{V}$ represent the relations in $\mathcal{D}$, and the edges in $\mathcal{E}$ represent foreign key references between two relations: there is an edge from $\mathrm{R}_\mathrm{j}$ to $\mathrm{R}_\mathrm{k}$ in $\mathcal{E}$ iff the primary key defined on $\mathrm{R}_\mathrm{k}$ is referenced by a foreign key defined in $\mathrm{R}_\mathrm{j}$. There can be multiple edges from $\mathrm{R}_\mathrm{j}$ to $\mathrm{R}_\mathrm{k}$ and we label each edge with the corresponding foreign key's attribute name. For simplicity, we omit edge labels in our examples and description if they are clear from the context.

In a relation $\mathrm{R}_\mathrm{i}$, we refer to a column as *text column* if its values are strings. Figure 11.1 shows an example database involving seven relations with a total of five text columns: Customer.CustName, Nation.NatName, Orders.Clerk, Part.PartName, and Supplier.SuppName. In the rest of this chapter, we focus only on text columns as well as primary or foreign key columns of relations.

### 11.2.2 Discovering Top-k PJ Queries by Example Spreadsheet

**Example spreadsheet.** An example spreadsheet is a multi-column table and serves as an interactive interface for PJ query discovery. Each cell of this spreadsheet is typed by the user, and could either be empty or contain some text. Figure 11.2(a) gives an example.

---

**Definition 3** *(Example spreadsheet)* An example spreadsheet $\mathrm{T}$ is a table with multiple rows $\{\mathrm{t}\}$ and columns $\mathrm{col}(\mathrm{T})$. Each row $\mathrm{t} \in \mathrm{T}$ is called an example tuple, where each cell is either a string (i.e., one or more terms) or empty. Let $\mathrm{t}[\mathrm{i}]$ denote its cell value on the column $\mathrm{i} \in \mathrm{col}(\mathrm{S})$, and let $\mathrm{t}[\mathrm{i}] = \emptyset$ if $\mathrm{t}[\mathrm{i}]$ is empty. Each row $\mathrm{t}$ contains at least one term and so does each column $\mathrm{T}[\mathrm{i}]$.

---

**Project-Join (PJ) queries.** We aim to discover queries in directed-tree shapes with projections and foreign key joins that generate a table from $\mathcal{D}$ to expand the user-given example spreadsheet $\mathrm{T}$.

---

**Definition 4** *(Project-Join Queries)* A project-join query $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ w.r.t to an example spreadsheet $\mathrm{T}$ is specified by:

- a *join tree* $\mathcal{J} \subseteq \mathcal{G}$, i.e., a directed subtree of the schema graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of the database $\mathcal{D}$ representing all the relations (vertices of $\mathcal{J}$) and joins (edges of $\mathcal{J}$) involved in the query – let $\mathrm{col}(\mathcal{J})$ be the set of all columns of relations in $\mathcal{J}$,
- a set of *projection columns* $\mathcal{C} \subseteq \mathrm{col}(\mathcal{J})$ from the relations in $\mathcal{J}$, which the join result is projected onto, and
- a *column mapping* $\varphi : \mathrm{col}(\mathrm{T}) \to \mathcal{C}$ from columns of the example spreadsheet $\mathrm{T}$ to the projection columns in $\mathcal{C}$ – it is a surjective function, i.e., $\forall \mathrm{c} \in \mathcal{C} : \exists \mathrm{i} \in \mathrm{col}(\mathrm{T})$ s.t. $\varphi(\mathrm{i}) = \mathrm{c}$.

---

It is important to ensure that there is no redundant table or projection column in the discovered PJ queries. Intuitively, a table or a projection column is redundant if, after it is dropped from the PJ query, the output relation matches the example spreadsheet equally well or even better. We formally define them as *minimal PJ queries* and only consider them as the candidates to be discovered.

---

**Definition 5** *(Minimal Project-Join Queries)* A PJ query $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ w.r.t. an example spreadsheet $\mathrm{T}$ is *minimal* iff

i) for any degree-1 vertex (relation) $\mathrm{R}$ in $\mathcal{J}$, there is a column $\mathrm{i} \in \mathrm{col}(\mathrm{T})$ s.t. $\varphi(\mathrm{i}) \in \mathrm{col}(\mathrm{R})$, i.e., every degree-1 relation $\mathrm{R}$ has a column of the example spreadsheet mapped to it and

ii) for every column $\mathrm{i}$ of $\mathrm{T}$ which is mapped to column $\mathrm{R}[\mathrm{j}]$ of a relation $\mathrm{R}$ in $\mathcal{J}$ through $\varphi$ (i.e., $\varphi(\mathrm{i}) = \mathrm{R}[\mathrm{j}]$), there exists at least one term in column $\mathrm{T}[\mathrm{i}]$ appearing in column $\mathrm{R}[\mathrm{j}]$.

---

In the rest of this chapter, when we refer to *PJ queries*, we refer to minimal project-join queries. For the example database in Figure 11.1, three PJ queries and their output relations are shown in Figure 11.2(b).

Let $\mathcal{A}(\mathrm{Q})$ be the output relation when $\mathrm{Q}$ is executed on database $\mathcal{D}$: joins in $\mathcal{J}$ are executed first, and then the results are projected on columns $\mathcal{C}$. Columns of the example spreadsheet $\mathrm{T}$ are mapped to columns $\mathcal{C}$ of the output relation $\mathcal{A}(\mathrm{Q})$ according to $\varphi$.

Property i) in Definition 5 is similar to the *minimality* of *candidate networks* in keyword search literatures like [ACD02; HGP03; LLWZ07]. In our case, degree-1 relations not satisfying i) can be excluded from $\mathcal{J}$ s.t. we have no less distinct tuples in the output relation $\mathcal{A}(\mathrm{Q})$, because they have no column in the projection and the join tree is still valid after the removal of them.

Property ii) in Definition 5 says that a column $\mathrm{i}$ in the example spreadsheet $\mathrm{T}$ should not be mapped to a column $\mathrm{R[j]}$ in the projection $\mathcal{C}$ if none of the terms in the column $\mathrm{T[i]}$ appears in $\mathrm{R[j]}$ (and the corresponding column in $\mathcal{A}(\mathrm{Q})$). Intuitively, if the two columns $\mathrm{T[i]}$ and $\mathrm{R[j]}$ have no overlapping vocabularies, they are likely from two different domains so it is meaningless to map $\mathrm{T[i]}$ to $\mathrm{R[j]}$. In fact, we can drop the column $\mathrm{i}$ from $\mathrm{T}$ to get a smaller example spreadsheet $\mathrm{T}'$, and drop the column $\mathrm{R[j]}$ from the projection $\mathcal{C}$ and the mapping $\varphi$, denoting as $\mathcal{C}' = \mathcal{C} - \{\mathrm{R[j]}\}$ and $\varphi'$; in our scoring model, we can prove that the relevance score of $\mathrm{Q}' = (\mathcal{J}, \mathcal{C}', \varphi')$ w.r.t. $\mathrm{T}'$ is no less than the score of $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ w.r.t. $\mathrm{T}$.

Based on our scoring model introduced next in Section 11.2.3, we will show that we do not “lose” in score by looking only at the minimal PJ queries (Proposition 1). A bit more formally, for any non-minimal PJ query $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$, we can find a minimal PJ query $\mathrm{Q}' = (\mathcal{J}', \mathcal{C}', \varphi')$ with $\mathcal{J}'$ as a subtree of $\mathcal{J}$ and/or $\varphi'$ as a sub-mapping of $\varphi$ such that the score of $\mathrm{Q}'$ is no less than the score of $\mathrm{Q}$. For example, consider the example spreadsheet in Figure 11.2(a) *without* column C, Figure 11.2(b)-(i) is no longer a minimal PJ query, as the degree-1 relation Part violates property i) and removing it will not reduce the score. Another example in Figure 11.2 is that it does not make sense to map column A in the example spreadsheet to column Nation.NName because any PJ query with this mapping violates property ii), and we can remove this pair of columns from the example spreadsheet/PJ query without reducing the score.

**End-to-end system task.** For a user-given example spreadsheet $\mathrm{T}$, the goal of our system is *to find minimal PJ queries with the top-*$\mathrm{k}$ *highest scores w.r.t.* $\mathrm{T}$. The incremental version of our task is: suppose we have found the top-$\mathrm{k}$ PJ queries for a user-given example spreadsheet $\mathrm{T}$, after one or more cells in $\mathrm{T}$ are updated by the user, how to find the updated top-$\mathrm{k}$ PJ queries efficiently.

## 11.2.3 Scoring Model for PJ Queries

The score of a PJ query $\mathrm{Q}$ w.r.t. an example spreadsheet $\mathrm{T}$ quantifies how well the $\mathrm{Q}$'s output $\mathcal{A}(\mathrm{Q})$ contains rows in $\mathrm{T}$. We first present the scoring model and then show how it allows us to tolerate relationship and domain errors for performing relevance ranking.

An IR system computes a score of a document w.r.t. a keyword query, which quantifies how well the former contains the terms in the latter. A straightforward way to compute the score of $\mathrm{Q}$ w.r.t. $\mathrm{T}$ is to treat $\mathrm{T}$ as a "*query*" (by concatenating all text in $\mathrm{T}$) and $\mathcal{A}(\mathrm{Q})$ as a "*document*" (again, by concatenating) and apply a traditional IR relevance scoring model [Sin01]. We do not adopt this model as we need to quantify how well $\mathcal{A}(\mathrm{Q})$ contains each example tuple with their columns aligned according to the mapping $\varphi$; it is difficult to do so in this model as it removes the row/column boundaries.

**Containment score w.r.t. single example tuple.** We first define a score $\mathsf{score}(\mathrm{t} \mid \mathcal{A}(\mathrm{Q}))$ to quantify how well $\mathcal{A}(\mathrm{Q})$ contains *a single example tuple* $\mathrm{t} \in \mathrm{T}$. Let $\mathsf{score}(\mathrm{t} \mid \mathrm{r})$ denote the similarity between an example tuple $\mathrm{t} \in \mathrm{T}$ and a row $\mathrm{r} \in \mathcal{A}(\mathrm{Q})$ in the PJ query output (referred to as *row-row similarity*). By definition of containment, $\mathsf{score}(\mathrm{t} \mid \mathcal{A}(\mathrm{Q}))$ should be high as long as there is one row $\mathrm{r} \in \mathcal{A}(\mathrm{Q})$ in the PJ query's output relation with a high row-row similarity $\mathsf{score}(\mathrm{t} \mid \mathrm{r})$ with $\mathrm{t}$; so we refer to the *most similar tuple* for $\mathrm{t}$ to define the containment score:

$$\mathsf{score}(\mathrm{t} \mid \mathrm{Q}) = \max_{\mathrm{r} \in \mathcal{A}(\mathrm{Q})} \mathsf{score}(\mathrm{t} \mid \mathrm{r}). \tag{11.1}$$

**Row-row similarity.** One way to get row-row similarity $\mathsf{score}(\mathrm{t} \mid \mathrm{r})$ between an example tuple $\mathrm{t} \in \mathrm{T}$ and a row $\mathrm{r} \in \mathcal{A}(\mathrm{Q})$ in the PJ query output is to treat $\mathrm{t}$ as a "query" (by concatenating the terms in all cells in $\mathrm{t}$) and $\mathrm{r}$ as a "document" (again, by concatenating). Again, this model is not suitable as we need to respect the mapping $\varphi$ while computing the row-row similarity. We need to compare a cell $\mathrm{t[i]}$ with the cell $\mathrm{r[\varphi(i)]}$ it is mapped to. Let $\mathsf{score}_{\mathrm{cell}}(\mathrm{t[i]} \mid \mathrm{r[j]})$ denote the cell similarity between an example tuple cell $\mathrm{t[i]}$ and a cell $\mathrm{r[j]}$ in an output row. We compute the row-row similarity by summing up the cell similarities for all columns.

$$\mathsf{score}(\mathrm{t} \mid \mathrm{r}) = \sum_{\mathrm{i} \in \mathrm{col(T)}} \mathsf{score}_{\mathrm{cell}}(\mathrm{t[i]} \mid \mathrm{r[\varphi(i)]}). \tag{11.2}$$

We use a simple cell similarity $\mathsf{score}_{\mathrm{cell}}(\mathrm{t[i]} \mid \mathrm{r[j]})$ as: how many terms in $\mathrm{t[i]}$ appear in $\mathrm{r[j]}$ if $\mathrm{t[i]}$ is non-empty and $0$ otherwise. We discuss how to adapt a more complicated IR-style cell similarity to perform relevance ranking in Section 11.7.2.

**Row containment score w.r.t. entire example spreadsheet.** We are now ready to define the *row-wise containment score* to quantify how well $\mathcal{A}(\mathrm{Q})$ contains *all* the example tuples in $\mathrm{T}$, denoted as $\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q})$. The more individual tuples in the example spreadsheet $\mathcal{A}(\mathrm{Q})$ contains, the higher should be the final score. So, a natural way is to sum up containment scores for all the example tuples in the example spreadsheet:

$$\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q}) = \sum_{\mathrm{t} \in \mathrm{T}} \mathsf{score}(\mathrm{t} \mid \mathrm{Q}) = \sum_{\mathrm{t} \in \mathrm{T}} \max_{\mathrm{r} \in \mathcal{A}(\mathrm{Q})} \mathsf{score}(\mathrm{t} \mid \mathrm{r}). \tag{11.3}$$

**Example 16** We compute the score $\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q})$ of PJ query $\mathrm{Q}$ in Figure 11.2(b)-(iii) w.r.t. the example spreadsheet $\mathrm{T}$ in Figure 11.2(a). Recall that cell similarity $\mathsf{score}_{\mathrm{cell}}(\mathrm{t[i]} \mid \mathrm{r[j]})$ is how many terms in $\mathrm{t[i]}$ appear in $\mathrm{r[j]}$ if non-empty, and $0$ otherwise. For each row in $\mathrm{T}$, the most similar row in $\mathcal{A}(\mathrm{Q})$ is shaded with the same color (yellow, pink and green for the three rows). The single tuple containment scores are 2, 1, and 1 respectively. So, $\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q}) = 4$. Similarly, the score between the same example spreadsheet and PJ query in Figure 11.2(b)-(ii) is $2 + 1 + 2 = 5$.

**Tolerating errors.** Naturally, the scoring function should have the following property: *higher the number of errors in the example spreadsheet with respect to the output of a*

*PJ query* $\mathrm{Q}$*, lower the score of* $\mathrm{Q}$. The above score $\mathsf{score}_{\mathrm{row}}$ satisfies this property. For example, the example spreadsheet in Figure 11.2(a) has 2 errors in the output of PJ query in Figure 11.2(b)-(ii) (Rick and Julie in column A do not appear in column Supplier.SName for the first two example tuples), while it has 3 errors in the output of PJ query (iii) (one term missed for each example tuple). From Example 16, we see that (ii) has a higher score $\mathsf{score}_{\mathrm{row}}$ than (iii). However, it *penalizes relationship and domain errors equally*. Relationship errors are more common, so we want to penalize them less than domain errors. We next introduce column containment score for that purpose.

**Column containment score.** We define column containment score that penalizes *only domain errors*. Subsequently, we will put it together with the row containment score $\mathsf{score}_{\mathrm{row}}$ to penalize the two classes of errors differently. A cell in column $\mathrm{i} \in \mathrm{col}(\mathrm{T})$ in an example spreadsheet $\mathrm{T}$ has a domain error in a PJ query $\mathrm{Q}$ iff it has one term not occurring in the mapped column $\varphi(\mathrm{i})$ of the join tree $\mathcal{J}$ of $\mathrm{Q}$. The column-wise containment score that quantifies how well the cells in each column $\mathrm{i} \in \mathrm{col}(\mathrm{T})$ are contained in the mapped column $\varphi(\mathrm{i})$ will penalize only domain errors.

For each cell in the example spreadsheet $\mathrm{T}$, we first find the *most similar cell* in the mapped column $\varphi(\mathrm{i})$ of the join tree $\mathcal{J}$ of $\mathrm{Q}$. We sum up the similarities between cells paired in this way to obtain the *column-wise containment score*:

$$\mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q}) = \sum_{\mathrm{i} \in \mathrm{col}(\mathrm{T})} \sum_{\mathrm{t} \in \mathrm{T}} \max_{\mathrm{r} \in \mathcal{J}[\varphi(\mathrm{i})]} \mathsf{score}_{\mathrm{cell}}(\mathrm{t}[\mathrm{i}] \mid \mathrm{r}[\varphi(\mathrm{i})]), \tag{11.4}$$

where let $\mathcal{J}[\varphi(\mathrm{i})]$ be the relation $\mathrm{T}[\mathrm{i}]$ is mapped to in database $\mathcal{D}$.

**Example 17** We compute the column-wise containment score $\mathsf{score}_{\mathrm{col}}$ of the PJ query $\mathrm{Q}$ in Figure 11.2(b)-(ii) w.r.t. the example spreadsheet $\mathrm{T}$ in Figure 11.2(a). Column A in $\mathrm{T}$ is mapped to column Supplier.SuppName (in the database in Figure 11.1) through $\varphi$. Only one (Rick) out of the three terms in $\mathrm{T}.\mathrm{A}$ appears in Supplier.SuppName. For each of the other two columns in $\mathrm{T}$, both terms can be found in the corresponding column in the database. So $\mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q}) = 5$. Similarly, the column-wise containment score of the PJ query $\mathrm{Q}$ in Figure 11.2(b)-(iii) is $3+2+2=7$. In contrast to Example 16, now (iii) has a higher score than (ii) because (iii) has no domain errors while (ii) has 2 domain errors.

**Putting it together.** We obtain the final *relevance score* $\mathsf{score}(\mathrm{T} \mid \mathrm{Q})$ of a PJ query $\mathrm{Q}$ w.r.t. an example spreadsheet $\mathrm{T}$ by taking a linear combination of row-wise and column-wise containment scores. We introduce a parameter $0 \leq \alpha \leq 1$ to control the relative penalty of the two classes of errors. Similar to prior work, we also penalize PJ queries with larger join trees as the relationship among the mapped columns in $\mathcal{J}$ is looser. We penalize it with a factor of $1+\ln(1+\ln|\mathcal{J}|)$, where $|\mathcal{J}|$ is the number of relations in $\mathcal{J}$.

$$\mathsf{score}(\mathrm{T} \mid \mathrm{Q}) = \frac{\alpha \cdot \mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q}) + (1-\alpha) \cdot \mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q})}{1+\ln(1+\ln|\mathcal{J}|)}. \tag{11.5}$$

**Minimality and scores.** In Proposition 1 below, we show why we are interested in finding only minimal PJ queries (Definition 5).

**Proposition 1** *Consider a user-given example spreadsheet* $\mathrm{T}$ *and a PJ query* $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ *in a database* $\mathcal{D}$. *If property i) in Definition 5 is not satisfied, let* $\mathrm{R}$ *be a degree-1 relation in* $\mathcal{J}$ *with no column in* $\mathrm{T}$ *mapped to it. Define a smaller* $\mathcal{J}' = \mathcal{J} - \mathrm{R}$ *and* $\mathrm{Q}' = (\mathcal{J}', \mathcal{C}, \varphi)$. *We have* $\mathsf{score}(\mathrm{T} \mid \mathrm{Q}) \leq \mathsf{score}(\mathrm{T} \mid \mathrm{Q}')$.

*If property ii) is violated, let* $\mathrm{i}$ *be a column in* $\mathrm{T}$ *s.t. no term in* $\mathrm{T[i]}$ *appears in the database column* $\mathrm{R[j]}$ *it is mapped to. We can remove the column* $\mathrm{i}$ *from* $\mathrm{T}$ *to get a smaller example spreadsheet* $\mathrm{T}'$, *and remove the column* $\mathrm{R[j]}$ *from the projection* $\mathcal{C}$ *and the mapping* $\varphi$: *let* $\mathcal{C}' = \mathcal{C} - \{\mathrm{R[j]}\}$ *and* $\varphi'$ *be the range-restriction of* $\varphi$ *by* $\mathcal{C}'$, *i.e.,* $\varphi'(\mathrm{i}) = \varphi(\mathrm{i})$ *iff* $\varphi(\mathrm{i}) \in \mathcal{C}'$ *and undefined otherwise. Let* $\mathrm{Q}'' = (\mathcal{J}, \mathcal{C}', \varphi')$. *We have* $\mathsf{score}(\mathrm{T} \mid \mathrm{Q}) = \mathsf{score}(\mathrm{T}' \mid \mathrm{Q}'')$.

## 11.3 System Architecture

The S4 system architecture is depicted in Figure 11.4, with two major components: *offline index building* and *online top-k ranking*.

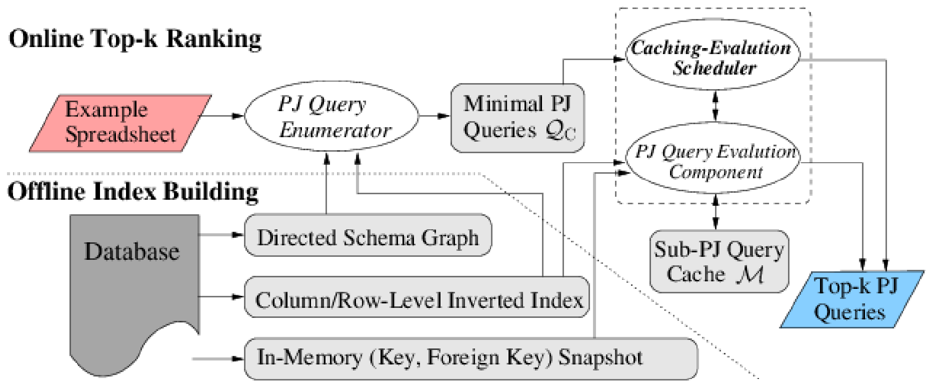

Online Top-k Ranking
Example Spreadsheet
PJ Query Enumerator
Minimal PJ Queries $\mathcal{Q}_C$
Caching-Evalution Scheduler
PJ Query Evalution Component
Sub-PJ Query Cache $\mathcal{M}$
Top-k PJ Queries
Offline Index Building
Database
Directed Schema Graph
Column/Row-Level Inverted Index
In-Memory (Key, Foreign Key) Snapshot

Figure 11.4: S4 System Architecture

## 11.3.1 Offline Index Building

**Directed Schema Graph.** First, we maintain the *directed schema graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in memory, which keeps schema-level information about the database $\mathcal{D}$, including names of relations/columns (in $\mathcal{V}$), and foreign keys (in $\mathcal{E}$).

Secondly, for the purpose of PJ query enumeration and evaluation (computing their scores), we build two types of in-memory indexes that are extensions to *inverted indexes* in traditional IR.

**Column-level inverted index.** Given a term $\mathsf{w}$, index $\mathsf{inv}(\mathsf{w})$ returns all the database columns where $\mathsf{w}$ appears in at least one row.

**Row-level inverted index.** For a term $\mathsf{w}$ and a column $\mathrm{R}[\mathrm{i}]$ in relation $\mathrm{R}$, $\mathsf{inv}(\mathsf{w}, \mathrm{R}[\mathrm{i}])$ returns all rows in $\mathrm{R}$, where $\mathsf{w}$ appears in column $\mathrm{i}$, and its term frequency in each cell.

Finally, for interactive performance, the PJ queries need to be executed and scored without accessing the database on disk using in-memory join indexes. Attivio Active Intelligence Engine uses a similar index to perform query-time join [SJTDP11]. We call it:

**In memory (key, foreign key) snapshot of the database.** We keep the primary-key and foreign-key columns of each row in each relation of the database materialized in memory.

### 11.3.2 Online Top-k Ranking

We adopt a *PJ-query enumeration-ranking* framework for online top-$k$ ranking. S4 processes a user-specified example spreadsheet in two steps as follows to output the top-$k$ PJ queries.

**PJ query enumeration.** The set of minimal PJ queries, denoted as $\mathcal{Q}_C$, for the example spreadsheet could all be candidate answers for the top-$k$. The component, *PJ Query Enumerator*, generates this set of PJ queries. We adapt the *CN generation* algorithm described in [HP02] for this component. Since the generation of $\mathcal{Q}_C$ (without computing their scores) can be done by accessing only the schema graph and column-level inverted index, it is quite efficient. Moreover, we compute an *upper bound* of its score for each PJ query in $\mathcal{Q}_C$. This upper bound can be also computed efficiently without executing any join. More details about the generation of $\mathcal{Q}_C$ and upper-bound score computation will be given in Section 11.4.1.

**PJ query ranking.** Taking PJ queries in $\mathcal{Q}_C$ as the input, the main contribution of this paper is about how to identify those with the top-$k$ highest scores from $\mathcal{Q}_C$. To this end, *PJ Query Evaluation Component* evaluates *some queries* in $\mathcal{Q}_C$ to get their scores. By *evaluating a PJ query*, we mean executing the query to compute its score w.r.t. the example spreadsheet. Because our scoring model quantifies how well the example spreadsheet is contained in the output of join and projection of a PJ query, to get the row containment score, we need to at least examine rows which either completely or partially contain the example tuples in the output. This process requires to execute joins and thus is the bottleneck in online top-$k$ ranking. Details about the evaluation of PJ queries will be given in Section 11.4.1 (basic version) and Section 11.5.1 (cache-aware version).

*Caching-Evaluation Scheduler* finds a strategy that specifies: i) which queries to be evaluated to get the top-$k$, ii) the order of evaluations, and iii) how to use the in-memory *Sub-PJ Query Cache* to speedup the evaluations. We focus on this scheduler in the rest part. We present a baseline strategy in Section 11.4 without utilizing the cache and a more efficient cache-aware strategy in Section 11.5.

## 11.4 Baseline Evaluation Strategy

In Section 11.4.1, we first introduce the basic operators in our evaluation strategy: how to enumerate all candidate PJ queries $\mathcal{Q}_C$ for an example spreadsheet and compute their upper-bound scores and exact scores. We also analyze their costs. For interactive speed, it is affordable to enumerate $\mathcal{Q}_C$ first and compute the upper-bound scores but it would be too expensive to compute the exact scores for all queries in $\mathcal{Q}_C$. To design more efficient approaches, in Section 11.4.2, we study what is the minimal set of queries we have to evaluate to get the top-$k$ given those upper bounds. We end this section with a worst-case optimal baseline strategy in Section 11.4.3.

### 11.4.1 Basic Operators in Evaluation Strategy

Before introducing our evaluation strategies, we give more details about the basic operators in our system and analyze their costs.

#### 11.4.1.1 Enumerating Minimal PJ Queries $\mathcal{Q}_C$

For a user-specified example spreadsheet $T$, we utilize the directed schema graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and column-level inverted index to generate $\mathcal{Q}_C$. For each column $T[i]$ in $T$, we first find all the columns $\mathcal{C}_i$ in $\mathcal{D}$ which contain at least one term in $T[i]$, called *candidate projection columns*. $\mathcal{C}_i$ is essentially the union of $\mathsf{inv}(w)$'s for all terms $w$'s in $T[i]$, i.e., $\mathcal{C}_i = \bigcup_{w \in T[i]} \mathsf{inv}(w)$.

Consider the example table in Figure 11.2. The candidate projection columns for column A are: Customer.CustName, Orders.Clerk (containing all the 3 terms in A) and Supplier.SuppName (containing only the first term, Rick). For column B, there is only one, Nation.NatName (containing both terms in B). And for column C, Part.PartName (contains both terms) is the only one.

Given candidate projection columns $\mathcal{C}_i$'s generated in the above process, to enumerate $\mathcal{Q}_C$, we can pick one column from each $\mathcal{C}_i$ to form $\mathcal{C}$ and mapping $\varphi$, and generate directed

Steiner trees to get $\mathcal{J}$ that connect to relations in $\mathcal{C}$, using the *CN generation* algorithm described in [HP02]. It is important to note that all PJ queries violating i) and ii) in Definition 5 are pruned during the enumeration.

#### 11.4.1.2 Computing Upper Bounds of Relevance Scores

It can be shown that the score of a PJ query $\mathrm{Q}$ w.r.t. $\mathrm{T}$, $\mathsf{score}(\mathrm{T} \mid \mathrm{Q})$, can be upper bounded by its column-wise score $\mathsf{score}_{\mathrm{col}}$ for any value parameter $\alpha$. This simple but effective upper bound can be computed in a light-weight way, without executing any join in $\mathrm{Q}$. We will use this upper bound frequently in the rest of this chapter.

**Proposition 2** (Upper Bound of Score) *For any* $0 \leq \alpha \leq 1$,

$$\mathsf{score}(\mathrm{T} \mid \mathrm{Q}) \leq \frac{\mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q})}{1 + \ln(1 + \ln |\mathcal{J}|)} \triangleq \overline{\mathsf{score}}(\mathrm{T} \mid \mathrm{Q}). \tag{11.6}$$

It suffices to show that $\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q}) \leq \mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q})$ to prove Proposition 2. The intuition is that, in $\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q})$, each row in $\mathrm{T}$ is matched to the most similar row in the output relation $\mathcal{A}(\mathrm{Q})$, while in $\mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q})$, each cell in $\mathrm{T}$ is matched to the most similar cell in the corresponding column of $\mathcal{A}(\mathrm{Q})$. The latter has a weaker constraint in matching so it produces a higher score.

**Computing $\overline{\mathsf{score}}$ and column containment score $\mathsf{score}_{\mathrm{col}}$.** To compute the upper bound $\overline{\mathsf{score}}$, it suffices to compute the column containment score $\mathsf{score}_{\mathrm{col}}$ in Equation (11.4). To compute $\mathsf{score}_{\mathrm{col}}$, for each term $\mathsf{w}$ appearing in column $\mathrm{i}$ of $\mathrm{T}$, suppose column $\mathrm{i}$ is mapped to column $\mathrm{j}$ of a relation $\mathrm{R}$, we need to scan $\mathsf{inv}(\mathsf{w}, \mathrm{R}[\mathrm{j}])$ once to compute the cell similarity $\mathsf{score}(\mathrm{t}[\mathrm{i}] \mid \mathrm{r}[\mathrm{j}])$ for each cell $\mathrm{t}[\mathrm{i}]$ in $\mathrm{T}$ and each cell $\mathrm{r}[\mathrm{j}]$ in $\mathrm{R}$. Then $\mathsf{score}_{\mathrm{col}}$ can be computed directly as in Equation (11.4). Refer to Algorithm 1 for more details.

**Proposition 3** *Consider an example spreadsheet* $\mathrm{T}$ *and a PJ query* $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ *in a database. For each column* $\mathrm{i}$ *in* $\mathrm{T}$ *and each term* $\mathsf{w}$ *in the column* $\mathrm{T}[\mathrm{i}]$, *suppose* $\mathrm{i}$ *is mapped to a column* $\mathrm{R}[\mathrm{j}]$ *in a relation* $\mathrm{R}$ *of* $\mathcal{J}$*, let* $\mathrm{l}_{\mathsf{w}} = |\mathsf{inv}(\mathsf{w}, \mathrm{R}[\mathrm{j}])|$ *be the number of rows in* $\mathrm{R}$ *that contain* $\mathsf{w}$ *in the column* $\mathrm{R}[\mathrm{j}]$, *i.e., the size of row-level inverted index* $\mathsf{inv}(\mathsf{w}, \mathrm{R}[\mathrm{j}])$ *for term* $\mathsf{w}$ *in column* $\mathrm{R}[\mathrm{j}]$. *Algorithm 1 computes* $\overline{\mathsf{score}}$ *and* $\mathsf{score}_{\mathrm{col}}$ *in* $\mathrm{O}(\sum_{\mathsf{w} \in \mathrm{T}} \mathrm{l}_{\mathsf{w}})$ *time.*

**Algorithm 1** Computing $\mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q})$ and $\overline{\mathsf{score}}(\mathrm{T} \mid \mathrm{Q})$

```
1: Initialize cell similarities $\mathsf{score}_{\mathrm{cell}}$ to be $0$ for all entries.
2: For each column $\mathrm{i}$ of $\mathrm{T}$: it is mapped to $\mathrm{R[j]}$ of relation $\mathrm{R}$
3:   For each tuple $\mathrm{t} \in \mathrm{T}$ and each term $\mathsf{w}$ in $\mathrm{t[i]}$
4:     Retrive row-level inverted index $\mathsf{inv}(\mathsf{w}, \mathrm{R[j]})$;
5:     For each row $\mathrm{r} \in \mathsf{inv}(\mathsf{w}, \mathrm{R[j]})$: $\mathsf{score}_{\mathrm{cell}}(\mathrm{t[i]} \mid \mathrm{r[j]})$++.
6: Compute $\mathsf{score}_{\mathrm{col}}(\mathrm{T} \mid \mathrm{Q})$ from $\mathsf{score}_{\mathrm{cell}}$ as in Equation (11.4).
7: Compute $\overline{\mathsf{score}}(\mathrm{T} \mid \mathrm{Q})$ as in Equation (11.6).
```

#### 11.4.1.3 Evaluating PJ Queries for Relevance Scores

We now introduce how S4 evaluates PJ queries to compute the exact relevance score $\mathsf{score}(\mathrm{T} \mid \mathrm{Q})$ and analyze its performance. According to Equation (11.5), the only missing part is the row containment score $\mathsf{score}_{\mathrm{row}}(\mathrm{T} \mid \mathrm{Q})$.

**Computing $\mathsf{score}$ and row containment score $\mathsf{score}_{\mathrm{row}}$.** As in Equation (11.3), for each example tuple $\mathrm{t}$ in $\mathrm{T}$, we need to find the most similar row in the output relation $\mathcal{A}(\mathrm{Q})$. Indeed, we can first execute the PJ query $\mathrm{Q}$, for example, sending it as a SQL query to the database (as in [SCC$^+$14]), and then examine each row in the output relation $\mathcal{A}(\mathrm{Q})$. To utilize our in-memory indexes and compute the scores more efficiently, we design an execution plan for $\mathrm{Q}$ using hash joins. More details are in Section 11.8.1. Following we focus on analyzing its complexity.

**Proposition 4** *Consider an example spreadsheet* $\mathrm{T}$ *and a PJ query* $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ *in a database. For each term* $\mathsf{w}$ *in* $\mathrm{T}$*, let* $\mathrm{l}_{\mathsf{w}}$ *be defined as in Proposition 3. For each relation* $\mathrm{R}$ *in* $\mathcal{J}$*, let* $|\mathrm{R}|$ *be the number of rows and* $\mathrm{d}_{\mathcal{J}}(\mathrm{R})$ *be the degree of* $\mathrm{R}$ *in* $\mathcal{J}$*. We can compute* $\mathsf{score}(\mathrm{T} \mid \mathrm{Q})$ *in* $\mathrm{O}(\sum_{\mathrm{R} \in \mathcal{J}} |\mathrm{R}| \cdot \mathrm{d}_{\mathcal{J}}(\mathrm{R}) + \sum_{\mathsf{w} \in \mathrm{T}} \mathrm{l}_{\mathsf{w}})$ *time.*

#### 11.4.1.4 Cost Analysis

It is quite common to enumerate candidate database queries in previous keyword search literature [ACD02; HP02; HGP03; LLWZ07]. Similarly in our case, we only access the schema graph and column-level inverted index to enumerate $\mathcal{Q}_{\mathrm{C}}$ (no need to execute any actual join).

Now, comparing the complexity of computing upper bounds of scores (Proposition 3) with computing exact scores (Proposition 4), we find that the additional cost, $\mathrm{O}(\sum_{\mathrm{R}\in\mathcal{J}}|\mathrm{R}|\cdot \mathrm{d}_{\mathcal{J}}(\mathrm{R}))$, to compute the exact score is truly the bottleneck. In the worst case, this cost is proportional to the total number of rows in all relations involved in a PJ query. On the other hand, the cost, $\mathrm{O}(\sum_{\mathrm{w}\in\mathrm{T}}\mathrm{l}_{\mathrm{w}})$, to compute the upper bound is only proportional to the number of rows that contain keyword terms in projection columns of the PJ query.

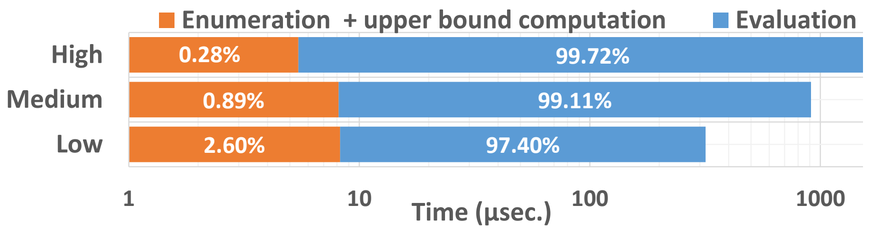


Figure 11.5: Average running time of "query enumeration + upper bound computation" v.s. "query evaluation" per PJ query.

Figure 11.5 compares i) the time for query enumeration plus upper bound computation (orange bars), with ii) the time to compute exact scores via query evaluation (blue bars), on average per query. We generate 50 example spreadsheets for CSUPP dataset and divide them into three buckets (H, M, L) based on the frequency of terms in the dataset (from highly frequent to lowly). Please refer to Section 11.6.1 for more details about the setting. All PJ queries are generated for those example spreadsheets, and for each, we compute both the upper-bound score and the exact score. The experimental result also shows that query enumeration plus upper bound computation requires a negligible fraction of the overall processing time. So, in the rest part, we assume that $\mathcal{Q}_{\mathrm{C}}$ *can be enumerated first for each given example spreadsheet* and $\overline{\mathsf{score}}$ *is associated with each query in* $\mathcal{Q}_{\mathrm{C}}$ *generated in the query enumeration step*.

### 11.4.2 Minimal Evaluation Set

From both theoretical and experimental analysis in the last subsection, we find that the bottleneck in our PJ-query enumeration-ranking framework is to *evaluate candidate PJ queries in* $\mathcal{Q}_{\mathrm{C}}$, i.e., to *execute PJ queries and compute their exact scores*; and on the other hand, we can enumerate all PJ queries in $\mathcal{Q}_{\mathrm{C}}$ and compute the upper bounds of scores for all of them quite efficiently. Now the major challenge translates to that of *evaluating as few PJ queries in* $\mathcal{Q}_{\mathrm{C}}$ *as possible to get the top-*$\mathrm{k}$ *with highest scores*, given that *an upper bound of score is associated with each one in* $\mathcal{Q}_{\mathrm{C}}$.

In the rest of this chapter, we will write $\overline{\mathsf{score}}(\mathrm{T} \mid \mathrm{Q})$ as $\overline{\mathsf{score}}(\mathrm{Q})$ and $\mathsf{score}(\mathrm{T} \mid \mathrm{Q})$ as $\mathsf{score}(\mathrm{Q})$ if $\mathrm{T}$ is clear from the context.

Given the set of queries in $\mathcal{Q}_{\mathrm{C}}$ and their upper-bound scores, we now analyze what is the minimal (sub)set $\mathcal{Q}_{\mathrm{min}} \subseteq \mathcal{Q}_{\mathrm{C}}$ of queries we have to evaluate to get the top-$\mathrm{k}$ with the highest scores in $\mathcal{Q}_{\mathrm{C}}$. Let $\mathcal{Q}_{\mathrm{C}} = \{\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_{\mathrm{N}}\}$, where $\mathrm{Q}_{\mathrm{i}}$'s are ordered by their upper-bound scores $\overline{\mathsf{score}}(\mathrm{Q}_1) \geq \ldots \geq \overline{\mathsf{score}}(\mathrm{Q}_{\mathrm{N}})$. Intuitively, if we evaluate queries in $\mathcal{Q}_{\mathrm{C}}$ in this order, we can *terminate* if

$$\mathrm{top}_{\mathrm{k}}\{\mathsf{score}(\mathrm{Q}_1), \ldots, \mathsf{score}(\mathrm{Q}_{\mathrm{i}})\} > \overline{\mathsf{score}}(\mathrm{Q}_{\mathrm{i}+1}), \tag{11.7}$$

where $\mathrm{top}_{\mathrm{k}}\{\ldots\}$ is the $\mathrm{k}$-th largest number in the set, we can assert that the top-$\mathrm{k}$ queries are among $\{\mathrm{Q}_1, \ldots, \mathrm{Q}_{\mathrm{i}}\}$. Let $\mathrm{i}^*$ be the minimal index $\mathrm{i}$ that satisfies the *termination condition* in Equation (11.7), let

$$\mathcal{Q}_{\mathrm{min}} = \{\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_{\mathrm{i}^*}\} \subseteq \mathcal{Q}_{\mathrm{C}}. \tag{11.8}$$

We can show that, informally, if based on only the upper-bound information $\overline{\mathsf{score}}$, $\mathcal{Q}_{\mathrm{min}}$, called *minimal evaluation set*, is the minimal subset of queries in $\mathcal{Q}_{\mathrm{C}}$ we have to evaluate to get the top-$\mathrm{k}$.

**Proposition 5** (Optimality of $\mathcal{Q}_{\mathrm{min}}$) *Given a set of PJ queries* $\mathcal{Q}_{\mathrm{C}}$ *and upper bounds* $\overline{\mathsf{score}}$ *of their scores. Any multi-step ranking algorithm to find queries with the top-*$\mathrm{k}$ *scores in* $\mathcal{Q}_{\mathrm{C}}$ *has to evaluate all queries in the set* $\mathcal{Q}_{\mathrm{min}}$ *(in Equation* (11.8)*) to compute their scores.*

The formalization of the class of *multi-step ranking algorithms* and the proof of this proposition are given in Section 11.9.

### 11.4.3 Worst-Case Optimal Baseline Strategy

Based on the upper bounds of scores and the optimality of $\mathcal{Q}_{\rm min}$ introduced in the above two subsections, we have a simple but "worst-case optimal" strategy BASELINE (Algorithm 2).

The idea is to evaluate queries $\rm Q_1, Q_2, \ldots$ in $\mathcal{Q}_{\rm C}$ one by one (to get the exact scores) in the descending order of upper-bound scores. Recall that upper-bound scores (line 1) can be computed efficiently and are associated with all queries in $\mathcal{Q}_{\rm C}$, as discussed in Sections 11.4.1.2 and 11.4.1.4. For each $\rm Q_i$, we use the operator introduced in Section 11.4.1.3 to compute its true score (line 4). If after we finish evaluating $\rm Q_i$ Equation (11.7) is satisfied, then we can terminate and output the current top-$\rm k$ among the evaluated PJ queries.

**Algorithm 2** Baseline Evaluation Strategy: BASELINE

```
Input:   queries in $\mathcal{Q}_{\rm C}$ and upper bounds of their scores
Output:  top-$\rm k$ PJ queries in $\mathcal{Q}_{\rm C}$ with the highest scores
1: Sort queries in $\mathcal{Q}_{\rm C}$: $\overline{\mathsf{score}}(\rm Q_1) \geq \ldots \geq \overline{\mathsf{score}}(\rm Q_N)$.
2: Initialize $\mathcal{Q}_{\rm topk} \leftarrow \emptyset$.
3: For $\rm i = 1$ to $\rm N$ do
4:     Evaluate $\rm Q_i$ to compute $\mathsf{score}(\rm Q_i)$.
5:     Let $\mathcal{Q}_{\rm topk} \leftarrow \mathcal{Q}_{\rm topk} \cup \{\rm Q_i\}$; if $|\mathcal{Q}_{\rm topk}| > \rm k$, keep only
       queries in $\mathcal{Q}_{\rm topk}$ with the top-$\rm k$ highest scores.
6:     If termination condition Equation (11.7) is satisfied, exit the loop.
7: Output $\mathcal{Q}_{\rm topk}$.
```

It is not hard to show that BASELINE evaluates only queries in $\mathcal{Q}_{\rm min}$ and thus is worst-case optimal for finding top-$\rm k$.

**Theorem 2** (Correctness and Optimality) BASELINE *correctly finds the PJ queries with the top-*$\rm k$ *scores among* $\mathcal{Q}_{\rm C}$ *and evaluates only queries in the minimal evaluation set* $\mathcal{Q}_{\rm min}$.

**Disadvantages of** BASELINE**.** BASELINE strategy in Algorithm 2 is worst-case optimal in terms of the number of PJ queries it evaluates. But the evaluation cost (or, response time) can be potentially improved significantly. BASELINE has several disadvantages. First, this baseline algorithm does not utilize frequent common subexpressions in $\mathcal{Q}_{\mathrm{C}}$. Indeed, we can cache the output relations of some subexpressions so that they can be re-used for more than one PJ query $\mathcal{Q}_{\mathrm{C}}$. Secondly, as we do not have infinite memory, to maximize the benefit we get from caching, we need to make the decisions of which subexpressions to be cached and when, for a batch of PJ queries. BASELINE examines queries in $\mathcal{Q}_{\mathrm{C}}$ one-by-one, so such decisions cannot be made wisely. The strategy we introduce in Section 11.5 improves BASELINE from the above two angles.

## 11.5 Optimizing Caching-Evaluation

We introduce our cache-aware optimization techniques in this section to overcome the disadvantages of BASELINE. The goal is still to find PJ queries with the top-$\mathrm{k}$ scores in $\mathcal{Q}_{\mathrm{C}}$ w.r.t. the user-given example spreadsheet $\mathrm{T}$. We will first discuss how to evaluate a PJ query. i.e., compute its score, when output relations of some sub-parts of it are cached, together with a cost model which quantify the cost of such cache-aware evaluation, in Section 11.5.1. We then formulate the *cache-evaluation scheduling problem*, i.e., an abstract version of the task to be solved by caching-evaluation scheduler in Section 11.5.2. Core technical challenges we resolve here are: to determine the order of PJ queries in $\mathcal{Q}_{\mathrm{C}}$ to be evaluated, which sub-PJ queries to be cached, and for how long, with the goal of minimizing the evaluation cost, or maximizing the benefit we get from caching. We show that it is NP-hard. Section 11.5.3 introduces a near-optimal solution to this problem. We will discuss how to extend our approach for incremental computation in Section 11.5.4.

### 11.5.1 Cache-Aware Evaluation of PJ Queries

Recall that the evaluation of $\mathsf{score}_{\mathrm{col}}$ is lightweight as we discussed in Section 11.4.1.2. So here, we focus on computing $\mathsf{score}_{\mathrm{row}}$ of a PJ query $\mathrm{Q}$ w.r.t. $\mathrm{T}$.

**Caching sub-PJ queries in evaluation.** Lets formally define a *sub-PJ query* of $\mathrm{Q}$, and discuss how to evaluate PJ query $\mathrm{Q}$ if the output relations of some sub-PJ queries of $\mathrm{Q}$ have been cached.

---

**Definition 6** *(Sub-PJ Query)* $\mathrm{Q}' = (\mathcal{J}', \mathcal{C}', \varphi_{\mathcal{C}'})$ is said to be a sub-PJ query of a PJ query $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$, iff $\mathcal{J}'$ is a subtree of $\mathcal{J}$; $\mathcal{C}' \subseteq \mathcal{C}$ is a subset of columns from relations in $\mathcal{J}'$ which the columns in $\mathrm{T}$ are mapped to; and $\varphi_{\mathcal{C}'}$ is a range-restriction of $\varphi$ by $\mathcal{C}'$, i.e., $\varphi_{\mathcal{C}'}(\mathrm{i}) = \varphi(\mathrm{i})$ iff $\varphi(\mathrm{i}) \in \mathcal{C}'$ and is undefined iff $\varphi(\mathrm{i}) \in \mathcal{C} - \mathcal{C}'$. We denote $\mathrm{Q}' \preceq \mathrm{Q}$ if $\mathrm{Q}'$ is a sub-PJ query of $\mathrm{Q}$.

Two types of subtrees are considered here: i) $\mathcal{J}'$ is a subtree of $\mathcal{J}$ rooted at some internal node, containing all leaves below; and ii) a type-i) subtree plus the parent of its root.

---

Figure 11.3 shows two sub-PJ queries $\mathrm{Q}'_1$ (left) and $\mathrm{Q}'_2$ (right) of the PJ query $\mathrm{Q}$ in Figure 11.2(b)-(i).

We have a *sub-PJ query cache* $\mathcal{M}$ in our system, which temporarily stores the output relations of some sub-PJ queries in a budgeted amount of memory. The execution plans of PJ-queries can be easily extended to take advantage of the cached output relations of sub-PJ queries: for a PJ query $\mathrm{Q}$ and a set of cached sub-PJ queries in $\mathcal{M}$, instead of starting from the leaves of $\mathrm{Q}$, we start from the output relations of maximal sub-PJ queries of $\mathrm{Q}$ in $\mathcal{M}$ and follow the execution plan of $\mathrm{Q}$ afterward. $\mathrm{Q}'$ is said to be a *maximal sub-PJ query of* $\mathrm{Q}$ *in* $\mathcal{M}$, iff $\mathrm{Q}' \preceq \mathrm{Q}$ and there is no $\mathrm{Q}''$ whose output relation is cached in $\mathcal{M}$ such that $\mathrm{Q}' \prec \mathrm{Q}'' \preceq \mathrm{Q}$. Intuitively, we want to utilize the output relations cached in $\mathcal{M}$ as much as possible. More details are given in Section 11.8.2.

**Cost model of evaluation.** We do not have unlimited memory to cache every single sub-PJ query. So our scheduling-evaluation scheduler needs to determine which sub-PJ queries to be cached and for how long, so as to maximize the benefit we obtain from caching, or equivalently, to minimize the total evaluation cost. To this end, a cost model needs to be introduced.

We define $\mathrm{cost}(\mathrm{Q})$ to be the cost of evaluating a (sub-)PJ query $\mathrm{Q}$, without utilizing the cache, and $\mathrm{cost}(\mathrm{Q}, \mathcal{M})$ be the cost of evaluating $\mathrm{Q}$ when a set of sub-PJ queries $\mathcal{M}$ have their output relations in the cache. We abuse the notation $\mathcal{M}$ a bit – we use it to denote both the set of sub-PJ queries as well as their cached output relations. Details about the cost model can be found in Equations (11.12) and (11.13) in Section 11.8.3, which is calibrated to the execution plans we use.

## 11.5.2 Caching-Evaluation Scheduling Problem

We are given a set of PJ queries $\mathcal{Q}_\mathrm{C} = \{\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_\mathrm{N}\}$, ordered by their upper-bound scores $\overline{\mathsf{score}}(\mathrm{Q}_1) \geq \overline{\mathsf{score}}(\mathrm{Q}_2) \geq \ldots \geq \overline{\mathsf{score}}(\mathrm{Q}_\mathrm{N})$. Let $\mathcal{T}(\mathrm{Q}_\mathrm{i})$ be the set of all *sub-PJ queries* of $\mathrm{Q}_\mathrm{i}$ and $\mathcal{T}(\mathcal{Q}) = \bigcup_{\mathrm{Q}_\mathrm{i} \in \mathcal{Q}} \mathcal{T}(\mathrm{Q}_\mathrm{i})$ for a set of PJ queries $\mathcal{Q} \subseteq \mathcal{Q}_\mathrm{C}$. We first introduce a general framework of our caching-evaluation strategies to find the top-$\mathrm{k}$ answers from $\mathcal{Q}_\mathrm{C}$, and then define the *caching-evaluation scheduling problem* to find the best strategy.

**Operators.** To utilize output relations of sub-PJ queries shared by multiple PJ-queries in $\mathcal{Q}_\mathrm{C}$, we maintain a cache $\mathcal{M}$ of size at most $\mathrm{B}$. Three types of operators are allowed:

a) $\underline{\mathrm{Evaluate}}(\mathrm{Q}, \mathcal{M})$: to evaluate a PJ or sub-PJ query $\mathrm{Q}$ using output relations cached in $\mathcal{M}$, and get its relevance score $\mathsf{score}(\mathrm{T} \mid \mathrm{Q})$ (for $\mathrm{Q} \in \mathcal{Q}_\mathrm{C}$) and output relation $\mathcal{A}(\mathrm{Q})$.

b) $\underline{\mathrm{Add}}(\mathrm{Q}, \mathcal{M})$: to store the output relation $\mathcal{A}(\mathrm{Q})$ in $\mathcal{M}$.

c) $\underline{\mathrm{Delete}}(\mathrm{Q}, \mathcal{M})$: to delete $\mathcal{A}(\mathrm{Q})$ from $\mathcal{M}$.

The size of $\mathcal{M}$, denoted as $|\mathcal{M}|$, is the memory available to keep the output relations of (sub-)PJ queries in $\mathcal{M}$. We want to ensure that, at any time, $|\mathcal{M}|$ could be at most $\mathrm{B}$.

**Caching-evaluation schedule and termination condition.** Initially, the cache $\mathcal{M}$ is empty, and for every PJ query $\mathrm{Q}_\mathrm{i} \in \mathcal{Q}_\mathrm{C}$ we only know its upper-bound score. We want to apply the above three types of operators in some order on PJ queries in $\mathcal{Q}_{\mathrm{C}}$ and their sub-PJ queries. The goal is that, at some point, the set of evaluated queries, denoted as $\mathcal{Q}_{\mathrm{E}}$, satisfies: $\mathcal{Q}_{\mathrm{E}} \supseteq \mathcal{Q}_{\min}$, i.e., the top-$\mathrm{k}$ have been found. Note that $\mathcal{Q}_{\min}$ is not known in advance.

**Objective.** Each type-a operator $\mathrm{Evaluate}(\mathrm{Q}, \mathcal{M})$ ($\mathrm{Q}$ is either a PJ query from $\mathcal{Q}_{\mathrm{C}}$ or a sub-PJ query) has a cost, $\mathrm{cost}(\mathrm{Q}, \mathcal{M})$, as defined in Equation (11.13) in Section 11.8.3. The goal is to minimize the total evaluation cost of type-a operators. Each type-b/c operator also has a cost but it is negligible compared to type-a's cost.

**Problem statement** (CACHE-EVAL SCHEDULER) *Given a set of PJ queries* $\mathcal{Q}_{\mathrm{C}} = \{\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_{\mathrm{N}}\}$*, with their upper-bound scores* $\overline{\mathsf{score}}(\mathrm{Q}_1) \geq \overline{\mathsf{score}}(\mathrm{Q}_2) \geq \ldots \geq \overline{\mathsf{score}}(\mathrm{Q}_{\mathrm{N}})$*, w.r.t. an example spreadsheet* $\mathrm{T}$*, a sequence of operators in type-a,b,c are chosen to be executed and the objective is to:*

$$\text{minimize} \sum_{\text{Op. } \mathrm{Evaluate}(\mathrm{Q},\mathcal{M})\text{'s executed}} \mathrm{cost}(\mathrm{Q}, \mathcal{M})$$

$$\text{s.t. } \mathcal{M} \text{ has size at most } \mathrm{B} \text{ at any time, and}$$

$$\text{eventually } \mathcal{Q}_{\mathrm{E}} \supseteq \mathcal{Q}_{\min}.$$

**Theorem 3** (Hardness) *Even when the set* $\mathcal{Q}_{\min}$ *is known, the* CACHE-EVAL SCHEDULER *problem is NP-complete, with* $|\mathcal{Q}_{\mathrm{C}}| + |\mathcal{T}(\mathcal{Q}_{\mathrm{C}})|$ *as the input size, where* $|\mathcal{T}(\mathcal{Q}_{\mathrm{C}})|$ *is the total number of sub-PJ queries of queries in* $\mathcal{Q}_{\mathrm{C}}$*.*

### 11.5.3 A Near-Optimal Strategy

We will introduce a near-optimal strategy, FASTTOPK, for the CACHE-EVAL SCHEDULER problem. It is based on two heuristics: *guessing the minimal evaluation set* and *caching critical sub-PJ queries*. We will also analyze the theoretical guarantee on performance it provides, in terms of the evaluation cost.

#### 11.5.3.1 Guessing The Minimal Evaluation Set

The first challenge is that the minimal evaluation set $\mathcal{Q}_{\mathrm{min}}$ is unknown to us. Our strategy BASELINE (Algorithm 2) examines only queries in $\mathcal{Q}_{\mathrm{min}}$ because it evaluates queries one by one, but $\mathcal{Q}_{\mathrm{min}}$ is known only after it terminates. Ideally, if we know $\mathcal{Q}_{\mathrm{min}}$ in advance, we can find *frequent* sub-PJ queries in $\mathcal{Q}_{\mathrm{min}}$ (the ones contained by many PJ queries), and cache their output relations so that they can be re-used multiple times in the evaluation. Indeed, we can consider all queries in $\mathcal{Q}_{\mathrm{C}}$ in one batch, but since $\mathcal{Q}_{\mathrm{min}}$ is usually a small subset of $\mathcal{Q}_{\mathrm{C}}$, sub-PJ queries that are frequent in $\mathcal{Q}_{\mathrm{C}}$ may not be frequent in $\mathcal{Q}_{\mathrm{min}}$ or even do not exist in $\mathcal{Q}_{\mathrm{min}}$ – so our decision of which sub-PJ queries to be cached based on their frequencies in $\mathcal{Q}_{\mathrm{C}}$ is likely to be sub-optimal.

We note that $\mathcal{Q}_{\mathrm{min}}$ is a “prefix” of $\mathcal{Q}_{\mathrm{C}}$, and can be uniquely specified by the index $\mathrm{i}^*$ as in Equations (11.7) and (11.8). So our heuristic to resolve this challenge is to create a few batches $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \ldots$ of queries in the order of $\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_\mathrm{i}, \ldots$ to approximate $\mathcal{Q}_{\mathrm{min}}$. We will optimize the cache-evaluation schedule for queries in each batch. After we finish evaluating each batch of queries in $\mathcal{B}_\mathrm{j}$, we check the termination condition Equation (11.7), and eventually we stop at $\mathcal{B}_{\mathrm{j}^*}$ after evaluating queries in $\mathcal{Q}_{\mathrm{E}} = \mathcal{B}_0 \cup \mathcal{B}_1 \cup \ldots \cup \mathcal{B}_{\mathrm{j}^*} \supseteq \mathcal{Q}_{\mathrm{min}}$. On one hand, we want to create as few batches as possible, i.e., $\mathrm{j}^*$ is small, so common sub-PJ queries across different queries can be found in one batch and we can cache and re-use their output relations; and on the other hand, we do not want to evaluate too many additional queries that are not in $\mathcal{Q}_{\mathrm{min}}$, i.e., $\mathcal{Q}_{\mathrm{E}} - \mathcal{Q}_{\mathrm{min}}$ is small.

The following batch-forming strategy balances the two concerns. The first batch of queries to be evaluated is $\mathcal{B}_0 = \{\mathrm{Q}_1, \ldots, \mathrm{Q}_\mathrm{k}\}$, as $\mathrm{Q}_\mathrm{i}$'s are ordered by the upper-bound scores and $\mathrm{B}_0$ has the least number of queries we have to evaluate to get the top-$\mathrm{k}$. After finishing evaluating this batch, if the termination condition in Equation (11.7) is not satisfied, we will consider the next batch with a slightly larger number of queries: $\mathcal{B}_1 = \{\mathrm{Q}_{\mathrm{k}+1}, \ldots, \mathrm{Q}_{\mathrm{k}(1+\varepsilon)}\}$. Again, if the termination condition is satisfied, we can stop and output the top-k; and otherwise, we consider the next batch. In general, the jth batch is $\mathcal{B}_\mathrm{j} = \{\mathrm{Q}_{\mathrm{k}(1+\varepsilon)^{\mathrm{j}-1}+1}, \ldots, \mathrm{Q}_{\mathrm{k}(1+\varepsilon)^\mathrm{j}}\}$, for some constant $\varepsilon > 0$.

**Algorithm 3** Near-Optimal Strategy: FASTTOPK

```
Input:  queries in $\mathcal{Q}_C$ and upper bounds of their scores
Output: top-k PJ queries in $\mathcal{Q}_C$ with the highest scores
1: Sort queries in $\mathcal{Q}_C$: $\overline{\mathsf{score}}(Q_1) \geq \ldots \geq \overline{\mathsf{score}}(Q_N)$.
2: Initialize $\mathcal{Q}_{\mathrm{topk}} \leftarrow \emptyset$.
3: Let the first batch be $\mathcal{B}_0 \leftarrow \{Q_1, \ldots, Q_k\}$, $i \leftarrow k$, and $j \leftarrow 0$.
4: Do
5:    $\mathrm{BatchEval}(\mathcal{B}_j)$ to get $\mathsf{score}(Q)$'s for all $Q \in \mathcal{B}_j$.
6:    Let $\mathcal{Q}_{\mathrm{topk}} \leftarrow \mathcal{Q}_{\mathrm{topk}} \cup \mathcal{B}_j$;
      keep only queries in $\mathcal{Q}_{\mathrm{topk}}$ with the top-$k$ highest scores.
7:    Let $i \leftarrow k(1+\varepsilon)^j$ and then $j \leftarrow j + 1$.
8:    Form next batch: $\mathcal{B}_j = \{Q_{i+1}, Q_{i+2}, \ldots, Q_{k(1+\varepsilon)^j}\}$.
9: While $(\mathrm{top}_k\{\mathsf{score}(Q_1), \ldots, \mathsf{score}(Q_i)\} \leq \overline{\mathsf{score}}(Q_{i+1}))$
10: Output $\mathcal{Q}_{\mathrm{topk}}$.
```

This high-level procedure is outlined in Algorithm 3. Recall the discussion in Section 11.4.1, upper-bound scores can be computed efficiently and are associated with all queries in $\mathcal{Q}_C$ (line 1). The loop (lines 4-9) forms batches one-by-one as described above. The value of $i$ in line 7 of every loop is the index of the last PJ query evaluated up to now, so line 9 can check the termination condition in Equation (11.7) to see whether the top-$k$ have been found.

Suppose the algorithm terminates with $i = i_{\mathrm{end}}$. In Section 11.5.3.3, we will utilize the fact that $i^* \leq i_{\mathrm{end}} \leq i^*(1+\varepsilon)$, where $i^* = |\mathcal{Q}_{\min}|$ is the least number of PJ queries any strategy has to evaluate, to give a performance guarantee of our strategy. Intuitively, based on this fact, we have that the size of $\mathcal{Q}_E$, the set of queries evaluated by FASTTOPK (Algorithm 3), is at most $(1+\varepsilon)|\mathcal{Q}_{\min}|$. So FASTTOPK does not examine too many additional queries that are not in $\mathcal{Q}_{\min}$. To bound the total evaluation cost, we can also show that there are at most $O(\log_{1+\varepsilon}(|\mathcal{Q}_{\min}|/k))$ batches.

The only missing building block in FASTTOPK (Algorithm 3) is now the subroutine $\mathrm{BatchEval}(\mathcal{B}_j)$ in line 5.

### 11.5.3.2 Caching Critical Sub-PJ Queries

Let's focus on the subroutine $\mathrm{BatchEval}(\mathcal{B}_j)$. We want to evaluate a batch $\mathcal{B}_j$ of queries using operators in type-a/b/c, so that the total cost is minimized. The basic idea of our approach for this subroutine is to partition queries in a batch $\mathcal{B}_j$ into groups, such that each group of PJ queries share at least one heavy-cost sub-PJ query, called *critical sub-PJ query*. The output relation of this sub-PJ query is cached in $\mathcal{M}$ if its size is no more than the budget $B$. We will later show that this seemingly simple heuristic has strong theoretical/practical performance guarantee.

**Critical sub-PJ query.** Let $\mathcal{T}(Q_i)$ be the set of all sub-PJ queries of $Q_i$, and $\mathcal{T}(\mathcal{B}_j) = \bigcup_{Q_i \in \mathcal{B}_j} \mathcal{T}(Q_i)$. A sub-PJ query $Q^* \in \mathcal{T}(Q_i)$ is said to be *critical* to $Q_i$ in $\mathcal{B}_j$, if i) $Q^*$ is shared by more than one query: there exists $Q_{i'} \in \mathcal{B}_j$ s.t. $i' \neq i$ and $Q^* \in \mathcal{T}(Q_{i'}) \cap \mathcal{T}(Q_i)$; and ii) $Q^*$ has the highest cost, $\mathrm{cost}(Q^*)$, among those in $\mathcal{T}(Q_i)$ satisfying condition i).

It is intuitive that the output relation of a critical sub-PJ query is worth caching in $\mathcal{M}$, because it can benefit the evaluation of at least one PJ query in $\mathcal{B}_j$, as condition i); and it has the highest cost among all such sub-PJ queries of a PJ query, as condition ii), so that we can benefit the most from caching it. We describe this heuristic for $\mathrm{BatchEval}(\mathcal{B}_j)$ more formally in Algorithm 4.

In line 1, all sub-PJ queries of $Q = (\mathcal{J}, \mathcal{C}, \varphi) \in \mathcal{B}_j$ can be enumerated efficiently by retrieving the rooted subtree at each node in $\mathcal{J}$. As we further elaborate in Section 11.8.3, the cost of each sub-PJ query $Q'_i$ can be computed efficiently from Equation (11.12) by summing up the sizes of relations in $Q'_i$ and row-level inverted indexes for columns in the projection of $Q'_i$. For each iteration (lines 2-10), the critical sub-PJ query $Q^*$ in $\mathcal{B}_j$ with the highest cost and output relation of size no more than $B$ is picked (line 4). Since we are focusing on foreign-key joins, we will use the number of rows in the root relation of $Q'$ multiplying the number of columns in the output relation as the size of the output

**Algorithm 4** $\mathrm{BatchEval}(\mathcal{B}_{\mathrm{j}})$: strategy for evaluating a batch of queries for their scores while minimizing the cost

```
Input:  queries in a batch $\mathcal{B}_{\mathrm{j}}$ and cache budget $\mathrm{B}$
Task:   get $\mathsf{score}(\mathrm{Q})$ by evaluating every $\mathrm{Q} \in \mathcal{B}_{\mathrm{j}}$
1:  Sort the $\mathrm{M}$ sub-PJ queries in $\mathcal{T}(\mathcal{B}_{\mathrm{j}})$ as:
        $\mathrm{cost}(\mathrm{Q}'_1) \geq \mathrm{cost}(\mathrm{Q}'_1) \geq \ldots \geq \mathrm{cost}(\mathrm{Q}'_{\mathrm{M}})$.
2:  While $\mathcal{B}_{\mathrm{j}}$ is not empty
3:    Clear $\mathcal{M}$ using type-c operators $\underline{\mathrm{Delete}}$.
4:    Pick $\mathrm{Q}^* = \mathrm{argmax}_{\mathrm{Q}' \in \mathcal{T}(\mathcal{B}_{\mathrm{j}})} \{\mathrm{cost}(\mathrm{Q}') \mid$
        $|\mathcal{A}(\mathrm{Q}')| \leq \mathrm{B} \ \wedge \ \exists \mathrm{i}_1 \neq \mathrm{i}_2 : \mathrm{Q}' \in \mathcal{T}(\mathrm{Q}_{\mathrm{i}_1}) \cap \mathcal{T}(\mathrm{Q}_{\mathrm{i}_2})\}$.
5:    If no such $\mathrm{Q}^*$ can be found, evaluate
        all the remaining queries in $\mathcal{B}_{\mathrm{j}}$ and terminate.
6:    Let $\mathrm{Critical}^{-1}(\mathrm{Q}^*) \leftarrow \{\mathrm{Q}_{\mathrm{i}} \in \mathcal{B}_{\mathrm{j}} \mid \mathrm{Q}^* \in \mathcal{T}(\mathrm{Q}_{\mathrm{i}})\}$.
7:    Execute $\underline{\mathrm{Evaluate}}(\mathrm{Q}^*, \mathcal{M})$ and $\underline{\mathrm{Add}}(\mathrm{Q}^*, \mathcal{M})$.
      (evaluate $\mathrm{Q}^*$ and store its output relation in $\mathcal{M}$)
8:    For each $\mathrm{Q}_{\mathrm{i}} \in \mathrm{Critical}^{-1}(\mathrm{Q}^*)$ do
9:      Evaluate $\mathrm{Q}_{\mathrm{i}}$ using $\mathcal{M}$: call $\underline{\mathrm{Evaluate}}(\mathrm{Q}_{\mathrm{i}}, \mathcal{M})$.
10:   Let $\mathcal{B}_{\mathrm{j}} \leftarrow \mathcal{B}_{\mathrm{j}} - \mathrm{Critical}^{-1}(\mathrm{Q}^*)$.
```

relation, $|\mathcal{A}(\mathrm{Q}')|$. The set of queries in $\mathcal{B}_{\mathrm{j}}$ containing $\mathrm{Q}^*$ as a sub-PJ query is put into a set $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$ is (line 6). We first evaluate and cache the output relation of $\mathrm{Q}^*$ in $\mathcal{M}$ (lines 7). Then all queries having $\mathrm{Q}^*$ as its sub-PJ query are evaluated to get their scores using $\mathcal{M}$ (lines 8-9), and removed form $\mathcal{B}_{\mathrm{j}}$ (line 10). After that, the cache $\mathcal{M}$ is cleared up (line 3).

**Example 18** Consider a batch of three PJ queries, $\mathcal{B}_{\mathrm{j}} = \{\mathrm{Q}_1, \mathrm{Q}_2, \mathrm{Q}_3\}$, where $\mathrm{Q}_1$-$\mathrm{Q}_3$ are depicted in Figure 11.2(b)-(i), (ii), (iii), respectively. Two sub-PJ queries $\mathrm{Q}'_1$ and $\mathrm{Q}'_2$ of them are shown in Figure 11.3 (left and right, respectively). Both are contained in two queries $\mathrm{Q}_1$ and $\mathrm{Q}_3$, so could potentially be critical sub-PJ queries to $\mathrm{Q}_1$ and $\mathrm{Q}_3$. If $\mathrm{Q}'_2$ has the largest cost $\mathrm{cost}(Q_2')$ among all such sub-PJ queries, then $Q_2'$ is a critical to both $Q_1$ and $Q_3$, and line 4 will pick $Q^* = Q_2'$. We have $\mathrm{Critical}^{-1}(Q^*) = \{Q_1, Q_3\} \subseteq B_j$ in line 6. Our strategy FASTTOPK would cache $\mathcal{A}(Q_2')$ in $\mathcal{M}$ first (line 7), and then use it to evaluate $Q_1$ and $Q_3$ (lines 8-9). After that, only $Q_2$ is left in $\mathcal{B}_j$ and will be evaluated as in line 5.

Although this subroutine is applied for each batch independently in Algorithm 3 to find the top-$k$ in $\mathcal{Q}_C$, we will later show that it has an overall performance guarantee, using the fact that the total number of batches is small (the last value of $j$ in Algorithm 3).

**Theorem 4** (Putting Together and Correctness) FASTTOPK *strategy (Algorithm 3 with subroutine* BatchEval *as Algorithm 4), correctly finds the queries in $\mathcal{Q}_C$ with the top-$k$ scores.*

### 11.5.3.3 Performance Analysis

We will start with the time complexity of our strategy FASTTOPK, and then analyze its approximation ratio.

**Time complexity.** The online response time of our system is determined by time spent i) on generating PJ queries in $\mathcal{Q}_C$ and their upper bounds, ii) on evaluating PJ queries, and iii) on optimizing the scheduling of caching and evaluations (FASTTOPK strategy). i) is negligible compared to ii) (analyzed in Section 11.4.1.4). ii) is modeled as the objective in our CACHE-EVAL SCHEDULER problem and the goal of FASTTOPK is to reduce it as much as possible. The purpose of the following theorem is to show that the time spent by FASTTOPK on iii) is negligible compared to ii), excluding time spent on executing type-a/b/c operators.

**Theorem 5** (Time Complexity) *Given a set of PJ queries in $\mathcal{Q}_C$ and their sub-PJ queries $\mathcal{T}(\mathcal{Q}_C)$ with costs and upper-bound scores associated, the time complexity of* FASTTOPK *is* $O(N + M_{all}(s_{max} + \log M_{all}))$ *(excluding the running time of operators in type-a/b/c chosen by* FASTTOPK *to be run), where* $N = |\mathcal{Q}_C|$ *is the number of PJ queries in $\mathcal{Q}_C$,* $M_{all} = \sum_{Q_i \in \mathcal{Q}_C} |\mathcal{T}(Q_i)|$ *here is the total number of sub-PJ queries of queries in $\mathcal{Q}_C$, and*

$\mathrm{s_{max}}$ *is the max size of a join tree* $\mathcal{J}$ *of a PJ query in* $\mathcal{Q}_\mathrm{C}$*. Because of the definition of a sub-PJ query, we have* $\mathrm{M_{all}} = \Theta(\mathrm{s_{max}} \cdot \mathrm{N})$.

**Performance ratio.** Our FASTTOPK strategy (Algorithms 3 and 4) provides a feasible solution (a scheduling of operators in type a/b/c) to the CACHE-EVAL SCHEDULER problem. We now compare the cost of this strategy with the cost of the optimal solution (which is hard to be found as shown in Theorem 3).

For a set of PJ queries $\mathcal{Q} = \{\mathrm{Q_i}\}$, let $\mathrm{cost_{TOT}}(\mathcal{Q})$ be the total cost of evaluating queries in $\mathcal{Q}$ one by one without caching:

$$\mathrm{cost_{TOT}}(\mathcal{Q}) = \sum_{\mathrm{Q_i} \in \mathcal{Q}} \mathrm{cost}(\mathrm{Q_i}).$$

Let $\mathrm{cost_{OPT}}(\mathcal{Q})$ be the cost of evaluating all queries in $\mathcal{Q}$ using the optimal strategy in the CACHE-EVAL SCHEDULER problem.

Let $\mathrm{cost_{SOL}}(\mathcal{Q})$ be the cost of evaluating all queries in $\mathcal{Q}$ using our FASTTOPK strategy in Algorithms 3 and 4.

**Theorem 6** (Performance Ratio) *Given PJ queries in* $\mathcal{Q}_\mathrm{C}$ *with upper-bound scores associated, the strategy* FASTTOPK *in Algorithms 3 and 4 evaluates a set of PJ queries* $\mathcal{Q}_\mathrm{E}$ *s.t.*

$$|\mathcal{Q}_\mathrm{min}| \le |\mathcal{Q}_\mathrm{E}| \le (1+\varepsilon)|\mathcal{Q}_\mathrm{min}|, \tag{11.9}$$

*and the benefit from caching*

$$\begin{aligned} &\mathrm{cost_{TOT}}(\mathcal{Q}_\mathrm{E}) - \mathrm{cost_{SOL}}(\mathcal{Q}_\mathrm{E}) \\ \ge &\frac{1}{2\mathrm{c}^2}\left(\mathrm{cost_{TOT}}(\mathcal{Q}_\mathrm{E}) - \log_{1+\varepsilon}\left(\frac{|\mathcal{Q}_\mathrm{min}|}{\mathrm{k}}\right) \cdot \mathrm{cost_{OPT}}(\mathcal{Q}_\mathrm{E})\right) \end{aligned} \tag{11.10}$$

*where* $\mathrm{c}$ *is the number of columns in* $\mathrm{T}$.

Informally, the above theorem gives guarantees for our FASTTOPK strategy from two aspects: i) it does not evaluate too many additional PJ queries in additional to the necessary ones in $\mathcal{Q}_\mathrm{min}$, as in Equation (11.9); and ii) $\mathrm{cost_{TOT}}(\mathcal{Q}_\mathrm{E}) - \mathrm{cost_{SOL}}(\mathcal{Q}_\mathrm{E})$ is the benefit we obtained from caching, and is lower bounded by the gap between the total cost and the cost of the optimal strategy (RHS of Equation (11.10)).

#### 11.5.3.4 Heuristics for Further Improvement

Our strategy can be further improved. Although not improving the performance ratio in Theorem 6, the following two heuristics are effective to improve its performance in practice.

The first heuristic is as follows. Consider each iteration of lines 3-10 in Algorithm 4, only one output relation (the one of $\mathrm{Q}^*$) is cached in $\mathcal{M}$. Indeed, if there is still room in $\mathcal{M}$, it will always be beneficial to cache more sub-PJ queries to speed up the evaluation of queries in $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$. So our heuristic here is to order queries in $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$ in such a way that "similar" queries (sharing common sub-PJ queries) are consecutive. While we evaluate these queries one-by-one in this order, the standard LRU replacement algorithm is applied to insert and replace output relations of sub-PJ queries in $\mathcal{M}$ – but note that we never replace the output relation of $\mathrm{Q}^*$ until finishing evaluating all queries in $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$. Such an order can be formed as follows. Starting with any query in $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$, in each step, we pick the query, which shares the most sub-PJ queries with the last one but is not in the order yet, to be the next one in this order. Repeat until all queries are placed in the order.

The second heuristic is an extension to our termination condition in Equation (11.7). We call it the *skipping condition*. During the execution of FASTTOPK, we maintain the current $\mathrm{k}$th highest score for all the queries that have been evaluated. In line 9 of Algorithm 4, before we evaluate $\mathrm{Q_i}$, we first check whether its upper-bound score is higher than the current top-$\mathrm{k}$ score – if not, we can safely skip the evaluation of $\mathrm{Q_i}$. This heuristic is particularly powerful and necessary for the last batch of queries in Algorithm 3, as this batch is usually large and contains queries not in $\mathcal{Q}_{\min}$.

### 11.5.4 Incremental Computation

Our strategy can be easily extended for incremental computation. The incremental version of our end-to-end system task is: suppose we have found the top-$\mathrm{k}$ PJ queries for a user-given example spreadsheet $\mathrm{T}$, after one or more cells in $\mathrm{T}$ are updated – the updated example spreadsheet is denoted as $\mathrm{T}'$ – how to find the top-$\mathrm{k}$ PJ queries for $\mathrm{T}'$ by re-using the evaluation results for $\mathrm{T}$.

If the user adds/deletes a column in $\mathrm{T}$, we re-start and generate a completely new caching-evaluation schedule using FASTTOPK (Algorithms 3 and 4), because in this case, the set of PJ queries, $\mathcal{Q}_{\mathrm{C}}'$, to be evaluated for $\mathrm{T}'$ are different from $\mathcal{Q}_{\mathrm{C}}$ for $\mathrm{T}$.

We focus on speeding up the case when the set of columns in $\mathrm{T}$ are unchanged, but some rows are updated (with one or more cells). In this case, $\mathcal{Q}_{\mathrm{C}}'$ may have large overlap with $\mathcal{Q}_{\mathrm{C}}$ and thus evaluation results for $\mathcal{Q}_{\mathrm{C}}$ can be re-used. The basic ideas are to derive a tighter upper bound of $\mathsf{score}(\mathrm{T}' \mid \mathrm{Q})$ based on the unchanged part of $\mathrm{T}$ and to schedule the incremental part of evaluation for $\mathcal{Q}_{\mathrm{C}}'$ carefully. Refer to Section 11.7.1 for more details.

## 11.6 Experimental Evaluation

We present an experimental study of the techniques proposed in this paper. We evaluate and compare three algorithms.

- NAIVE: evaluates all the enumerated PJ queries in $\mathcal{Q}_{\mathrm{C}}$;
- BASELINE (Algorithm 2): as described in Section 11.4.3;
- FASTTOPK (Algorithms 3-4): as described in Section 11.5.

We compare the performance of the three algorithms, and evaluate their sensitivity with respect to various parameters (Section 11.6.2). We also conduct a user study to evaluate the effectiveness of our scoring model (Section 11.6.3). Additional experiments about incremental computation are deferred to the appendix.

### 11.6.1 Settings of Experiments

We have implemented all the algorithms using C++/CLI (Common Language Infrastructure) on a Windows 8.1 machine with an Intel i7-4770 CPU at 3.4GHz with 16GB RAM.

**Datasets.** We use two datasets to evaluate the system performance: CSUPP and AdventureWorks. Our primary dataset, CSUPP, is a real-life database containing information related to customer service and IT support from a Fortune 500 Company. It has a size of 95GB. AdventureWorks, ADVW for short, is a synthetic database with information related to sales, purchasing, product management and contact management with size 300MB [Adv]. Although ADVW has a small size, we use it for its realistic and complex schema (93 primary key-foreign key edges compared with 63 in CSUPP), and also scale up its dimension/fact tables by creating new rows.

| | # Relations | # Columns | # Text Columns | # Edges |
|---|---|---|---|---|
| CSUPP | 105 | 1721 | 821 | 63 |
| ADVW | 71 | 650 | 104 | 93 |

For the user study, we use the real database IMDB [IMD] with information about movies, because our judges are more familiar with the movie domain compared with CSUPP or ADVW.

**Index building.** To build the inverted indexes, we tokenize each cell in each text column in the database. We discard tokens containing non-alphanumeric characters and those with more than 15 characters. For each token, we construct a list consisting of column identifiers (which uniquely identifies a column across all columns in the database) of all columns containing it. This forms the *column-level inverted index*. For each token in each column, we construct a list consisting of the row identifiers (which identifies the row within the relation) of all cells containing it in the column. This forms *the row-level inverted index*. We also build an *in-memory (key, foreign key) snapshot* as discussed in Section 11.3.1. We store all indexes in memory. Table 11.1 shows the index sizes. For both databases, the total index size is about **7%** of the database size.

| | Inv. index (MiB) | (key,fk) snap. (MiB) | Tokens |
|---|---|---|---|
| CSUPP | 4759.7 | 1237.4 | 6434684 |
| ADVW | 6.86 | 12.57 | 125083 |

Table 11.1: Index sizes

**Example spreadsheet (ES) generation.** We manually choose 10 semantically meaningful join queries with 6 or more text columns. We execute them and project the results on all the text columns involved. We generate an ES with $m$ rows and $n$ columns by (i) randomly choosing one of the semantically meaningful join queries and (ii) randomly choosing $m$ rows and $n$ columns from its output. We keep only the first token of the cell and all cells of the ES are non-empty. We use $m = 3$ and $n = 3$ in all our experiments.

To simulate real-life inputs, we introduce relationship errors in the ESs (default is 2 errors). To introduce a relationship error, we randomly select a cell of an ES generated above and replace it with the value of another cell in the same column in the join query's output. As before, we keep only the first token of the chosen cell.

We generate 50 ESs for CSUPP and 450 ESs for ADVW. We divide the 50 ESs for CSUPP into 3 buckets, namely *low*, *medium*, and *high*, based on the sizes of row-level inverted indexes of terms in the ESs (from lowly frequent to highly frequent). There are 25, 15, and 10 ESs in the three buckets, respectively. This is to test how our approaches are sensitive about the frequency of terms.

| Description | Symbol | Ranges and default values |
|---|---|---|
| Param. in scoring model | $\alpha$ | $0.5, 0.6, 0.7, \underline{0.8}, 0.9, 1.0$ |
| k in top-k | $k$ | $5, 10, 20, \underline{50}, 100$ |
| Batch increase factor | $\varepsilon$ | $0.2, \underline{0.4}, 0.6, 0.8, 1.0, 2.0$ |
| Cache size (MiB) | $B$ | 100, 200, 500, <u>1000</u>, 2000 |
| # relationship errors | $e$ | $0, 1, \underline{2}, \ldots, (m-1) * n$ |

Table 11.2: Parameters we vary in our experiments along with their description, value ranges, and default values (underlined)

### 11.6.2 System Performance

We compare the algorithms by: (i) execution time and (ii) number of PJ query-row evaluations (times PJ queries are evaluated on rows in the ES). (ii) indicates the benefit of *using upper bounds* $\overline{\text{score}}$ *of scores for early termination*. (i) indicates the combined benefit of *caching shared sub-PJ queries* and *early termination*.

In our experiments, we vary the 5 parameters in Table 11.2. Unless otherwise specified, we use the underlined default values. We use CSUPP as the dataset in Exp-I to IV, and ADVW in Exp-IIV.

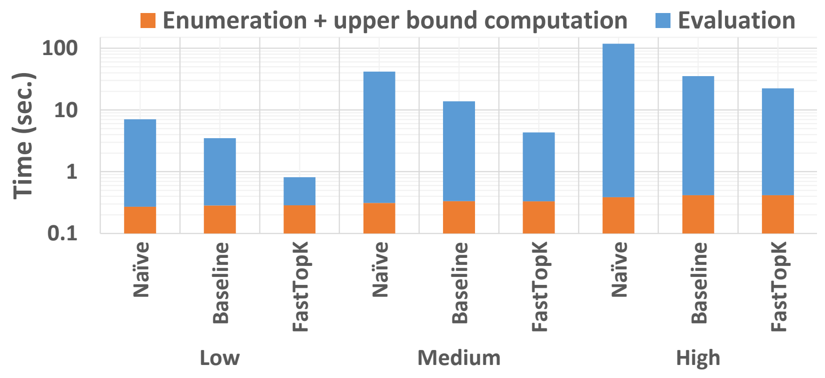


Figure 11.6: Comparison of FASTTOPK with NAIVE and BASELINE

**Exp-I: Comparing FASTTOPK with NAIVE and BASELINE.** Figure 11.6 shows the average execution times (in log scale) of the three algorithms for each of the three ES buckets (low, medium and high). The execution time here is partitioned into query "enumeration + upper bound computation" and "evaluation" (evaluating PJ queries to compute their scores). Figure 11.6 shows that the "enumeration + upper bound computation" part takes a tiny fraction of the overall execution time for all the three approaches.

We used default values of the 5 parameters shown in Table 11.2 for this experiment. FASTTOPK outperforms NAIVE *by factors of 11, 10 and 5* for the low, medium and high buckets respectively. NAIVE often takes several minutes to return answers, and it can be found from Figure 11.6 that the crucial bottleneck is query evaluation to compute scores. Such inefficiency motivates the problem addressed in this paper for interactive query discovery.

The improvement of execution time in BASELINE and FASTTOPK is the combined benefit gained from using the upper bounds $\overline{\text{score}}$ for early termination and caching shared sub-PJ queries. Without using the upper bounds, NAIVE has to go through and evaluate all the PJ queries enumerated in $\mathcal{Q}_C$; but with the help of the upper bounds, BASELINE and

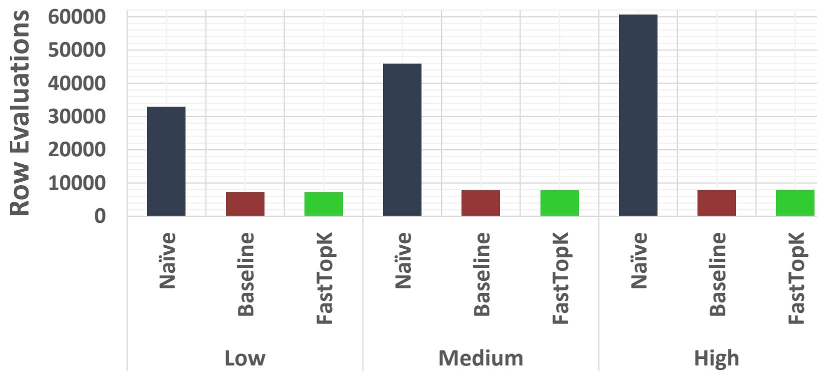
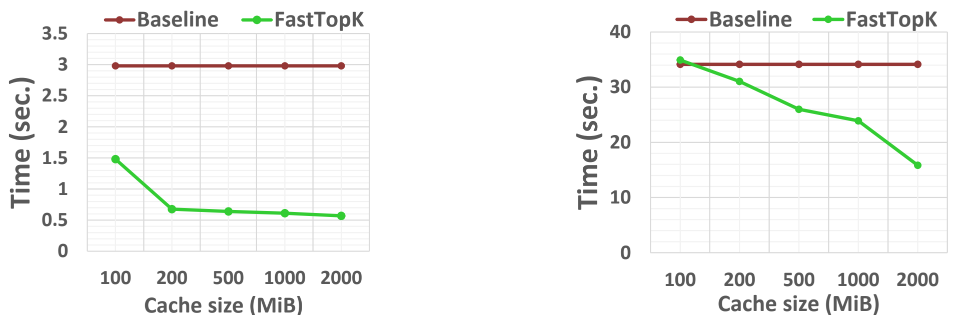


Figure 11.7: Amount of queries evaluated by NAIVE (without using upper bound $\overline{\text{score}}$), BASELINE (using $\overline{\text{score}}$), and FASTTOPK (using $\overline{\text{score}}$)

FASTTOPK can terminate as soon as the condition in Equation (11.7) is satisfied. Figure 11.7 plots the numbers of queries evaluated by the three approaches. The significantly smaller numbers of queries evaluated by BASELINE and FASTTOPK explain their faster execution time compared with NAIVE.

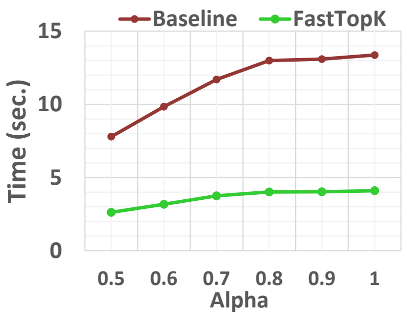
In Figure 11.6, FASTTOPK outperforms BASELINE *by factors of 5, 3 and 1.5* for the low, medium and high buckets respectively. This shows the benefit gained from solving the problem of *caching-evaluation scheduling* by our FASTTOPK strategy.



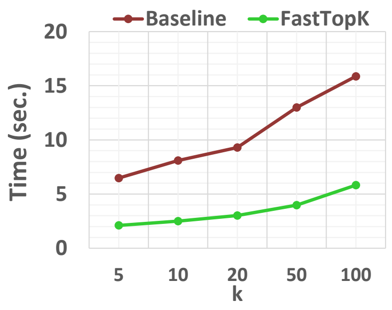
(a) Execution time for “low” bucket (b) Execution time for “high” bucket

Figure 11.8: Varying cache size B for ESs in “low”/“high” bucket

**Exp-II: Vary cache size.** Figure 11.8 shows execution time of the two algorithms for the low/high ES buckets. FASTTOPK outperforms the BASELINE for all cache sizes. Higher the cache size, more the sharing, larger the gap. With a cache size of $B = 2$GiB, FASTTOPK outperforms BASELINE *by a factor of 6X for the low-cost ESs* and *by 3.7X for medium-cost ESs*. The gap is smaller for high-cost ESs; FASTTOPK outperforms BASELINE *by a factor of 2.3X* for $B = 2$GiB. It is because the results of many of those common sub-PJ queries are too large to fit in the cache. We need a larger cache to obtain the full benefit of sharing for high-cost ESs and get speedups. The trend for the medium ES bucket is similar.



(a) Varying score weight $\alpha$



(b) Varying $k$

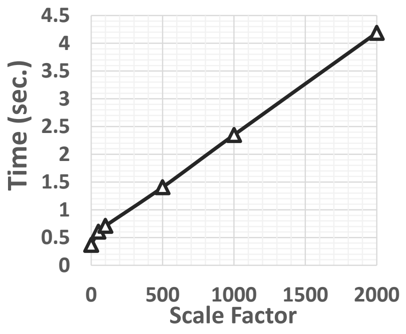
Figure 11.9: Varying $\alpha$ and $k$ for ESs in "medium" bucket

**Exp-III: Vary parameter $\alpha$.** Figure 11.9(a) shows the execution times of the two algorithms with different values of $\alpha$. Since $(1 - \alpha)$ is the weight on the column containment score $\mathsf{score}_{\mathrm{col}}$ and the upper bound score is proportional to $\mathsf{score}_{\mathrm{col}}$, smaller values of $\alpha$ imply tighter upper bound scores and thus faster early termination. Hence, the number of PJ query-row evaluations and the execution time of both algorithms increase in $\alpha$. FASTTOPK outperforms the BASELINE *by a factor of 3.5X* for all values of $\alpha$. While they evaluate almost the same number of PJ queries, FASTTOPK performs better due to caching shared outputs of common sub-PJ queries.

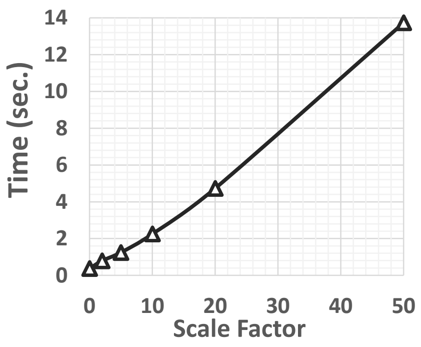
**Exp-IV: Vary $k$.** With increase in $k$, both approaches evaluate more PJ queries before they terminate, and thus need more execution time. They perform almost the same number of PJ Query-row evaluations. But, due to shared evaluation, FASTTOPK outperforms BASELINE *by a factor of 3-4X* for $k$'s as in Figure 11.9(b).

**Exp-V: Vary number of relationship errors.** We generate different sets of ESs with different numbers of relationship errors (varying from 0 to 5). Higher the number of errors, lower the final scores of the top-k PJ queries. A lower kth highest score delays the satisfaction of the termination condition. So the number of PJ query-row evaluations increase significantly with the number of errors. Overall, FASTTOPK outperforms BASELINE *by a factor of 2-6X.*

**Exp-VI: Vary** $\varepsilon$. We find that FASTTOPK is robust to $\varepsilon$. There is negligible change in execution time as we vary $\varepsilon$ from 0.2 to 2.0, so we omit the plot. One would expect the performance to suffer when $\varepsilon$ is high (say, 2.0) since FASTTOPK would evaluate many PJ queries outside the minimum evaluation set. However, due to both cache-evaluation scheduling and the skipping condition, it does not incur much more cost of evaluating such PJ queries.



(a) Scaling up dimension tables



(b) Scaling up fact tables

Figure 11.10: Varying scale factor in ADVW.

**Exp-VII: Scale up dimension tables and fact tables.** In this experiment, we start with the original ADVW database and scale it up to create databases with different statistical properties.

First, we scale up the dimension tables by creating new rows containing the same values as existing rows (but different row identifiers). We do not modify the fact tables, i.e., the new rows are not referenced by any fact rows. Figure 11.10(a) shows that the average execution time (over 450 ESs) of FASTTOPK increases slowly as we increase the scale factor (# new rows created for each existing row) from 1 to 2000. This is due to increase in cost of retrieving rows from the row-level inverted index. There is no increase in join cost (hash lookups) as the fact table is unchanged.

Next, we scale up the fact tables by creating new fact rows that reference the same dimension table rows as existing fact rows. We do not modify the dimension tables. This is to test the case that relations with a huge number of tuples and relatively few unique values in certain columns. Figure 11.10(b) shows the average execution time of FASTTOPK as we increase the scale factor (# new fact rows created for each existing fact row) from 1 to 50. The execution time increases at a much faster rate (superlinear) compared with the first case. This is due to increase in join cost (hash lookups), although the inverted index retrieval cost does not change. This also shows that the join cost dominates the overall cost of query processing.

## 11.6.3 User Study

We have conducted a user study in IMDB to evaluate the effectiveness of our scoring model. We use the IMDB database for this study as our judges are more familiar with the movie domain compared with CSUPP or ADVW. For the fairness of evaluation, we generate ESs from a source different from IMDB. We use HTML tables extracted from the web [CHZ$^+$08]. We select HTML tables about movies by creating a list of movies and checking for overlap of the subject column of the table with that list. We generate 52 ESs from randomly selected rows and columns of randomly selected movie HTML tables. For each ES, we compute the top-10 PJ queries and present it to the three judges via a web-based user interface.

We have all the human judges get familiar with and agree on the organization of the IMDB database, and give them access to each original HTML table, so that they can mark each PJ query as relevant or non-relevant to an ES. Different subsets of ESs are assigned to different judges. On average, judges marked 2.3 results as relevant per ES. The overall mean reciprocal rank (MRR)[1] is $0.79$; it shows that the relevant result(s) typically appear at the top.

To study the effectiveness for varying characteristics of the ES, we divide the 52 ESs into 3 buckets high (highly frequent terms), medium, and low, based on posting list sizes of the terms in the ESs. The MRRs for the high, medium and low buckets are $0.87$, $0.78$ and $0.71$, respectively. The MRR is quite stable across all the buckets; the MRR values are slightly lower for the low and medium buckets due to presence of a few ESs containing foreign language movies (which are not well-covered by IMDB) in these buckets.

## 11.7 Extension and Discussion

### 11.7.1 Incremental Computation

Let $\mathrm{T}' = \mathrm{T}^{\mathrm{old}} \cup \mathrm{T}^{\mathrm{new}}$, where $\mathrm{T}^{\mathrm{old}}$ is the set of unchanged rows that are identical to the ones in $\mathrm{T}$, and $\mathrm{T}^{\mathrm{new}}$ is the set of new rows or updated rows in $\mathrm{T}'$. Recall that $\mathcal{Q}_{\mathrm{E}} \subseteq \mathcal{Q}_{\mathrm{C}}$ is the set of PJ queries that have been evaluated for $\mathrm{T}$. For the new set of queries, $\mathcal{Q}_{\mathrm{C}}{}'$, generated by the PJ query enumerator for $\mathrm{T}'$, can be partitioned into $\mathcal{Q}_{\mathrm{C}}{}' = \mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}} \cup \mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}}$, where $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}} = \mathcal{Q}_{\mathrm{C}}{}' \cap \mathcal{Q}_{\mathrm{E}}$ and $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}} = \mathcal{Q}_{\mathrm{C}}{}' - \mathcal{Q}_{\mathrm{E}}$. For each query in $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}$, since it has been evaluated for $\mathrm{T}$, we keep and re-use its score w.r.t. rows in $\mathrm{T}^{\mathrm{old}}$; and for each query in $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}}$, since it is new or unevaluated before, we need to (re-)evaluate it for every row in $\mathrm{T}'$. We will discuss how to utilize the "partial scores" of queries in $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}$ to get better upper bounds of scores, and how to take the incremental changes into consideration when scheduling caching-evaluation.

**Improved upper-bound scores for $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}$.** For a query $\mathrm{Q} \in \mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}$, a better upper bound of its score can be computed as a combination of its old score w.r.t. rows in $\mathrm{T}^{\mathrm{old}}$ and

[1]Defined as $\frac{1}{\text{number of inputs}} \sum_{\text{each input}} \frac{1}{\text{rank of the first right answer}}$

column-wise score w.r.t. the rest part of $\mathrm{T}'$. For any $0 \leq \alpha \leq 1$, we have

$$\begin{aligned} \mathsf{score}(\mathrm{T}' \mid \mathrm{Q}) &\leq \mathsf{score}(\mathrm{T}^{\mathrm{old}} \mid \mathrm{Q}) + \frac{\mathsf{score}_{\mathrm{col}}(\mathrm{T}^{\mathrm{new}} \mid \mathrm{Q})}{1 + \ln(1 + \ln |\mathcal{J}|)} \\ &\leq \overline{\mathsf{score}}(\mathrm{T}' \mid \mathrm{Q}) \quad \text{(the one defined in Proposition 2).} \end{aligned} \tag{11.11}$$

Note that $\mathsf{score}(\mathrm{T}^{\mathrm{old}} \mid \mathrm{Q})$ in (11.11) is known, as Q has been evaluated for rows in $\mathrm{T}^{\mathrm{old}}$. $\mathsf{score}_{\mathrm{col}}$ can be computed as in Section 11.4.1.2. Compared to $\overline{\mathsf{score}}$, RHS of (11.11) is a better upper bound of score, and thus implies a smaller minimal evaluation set $\mathcal{Q}'_{\min}$ for $\mathrm{T}'$. So our strategies terminate more quickly using this upper bound.

**Incremental caching-evaluation scheduling.** We only need to modify our cost model so that the incremental updates can be automatically considered in our strategy FASTTOPK (Algorithms 3-4). In our current cost model (11.12)-(11.13) in 11.8.3, we need to evaluate each query $\mathrm{Q} = (\mathcal{J}, \mathcal{C}, \varphi)$ for every row in T. The new cost model to handle updates will take into consideration the number of rows in $\mathrm{T}'$ we need to evaluate Q on. i) For a (sub-)PJ query $\mathrm{Q} \in \mathcal{T}(\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}) - \mathcal{T}(\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}})$, we only need to evaluate it for rows in $\mathrm{T}^{\mathrm{new}}$, so we define $\mathrm{cost}'(\mathrm{Q}) = |\mathrm{T}^{\mathrm{new}}| \cdot \mathrm{cost}(\mathrm{Q})$ and $\mathrm{cost}'(\mathrm{Q}, \mathcal{M}) = |\mathrm{T}^{\mathrm{new}}| \cdot \mathrm{cost}(\mathrm{Q}, \mathcal{M})$; ii) for a (sub-)PJ query $\mathrm{Q} \in \mathcal{T}(\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}}) - \mathcal{T}(\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}})$, we need to evaluate it for all rows in $\mathrm{T}'$, so we define $\mathrm{cost}'(\mathrm{Q}) = |\mathrm{T}'| \cdot \mathrm{cost}(\mathrm{Q})$ and $\mathrm{cost}'(\mathrm{Q}, \mathcal{M}) = |\mathrm{T}'| \cdot \mathrm{cost}(\mathrm{Q}, \mathcal{M})$; and iii) for a (sub-)PJ query $\mathrm{Q} \in \mathcal{T}(\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}) \cap \mathcal{T}(\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}})$, we create two copies of it – the one belonging to queries in $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{old}}$ is associated with cost as i) and the one belong to queries in $\mathcal{Q}_{\mathrm{C}}{}^{\mathrm{new}}$ is associated with cost as ii). Such a weighted cost model is applied in FASTTOPK to get updated top-k PJ queries.

## Experimental Results

We generate complete $3 \times 3$ example spreadsheets in CSUPP as described in Section 11.6.1. We simulate incremental input by starting with the completely filled-out first row and then adding one cell at a time from the complete example spreadsheet (i.e., 6 cell additions). Our simulator adds cells row-by-row, from left to right. We average our results over the 50 example spreadsheets.

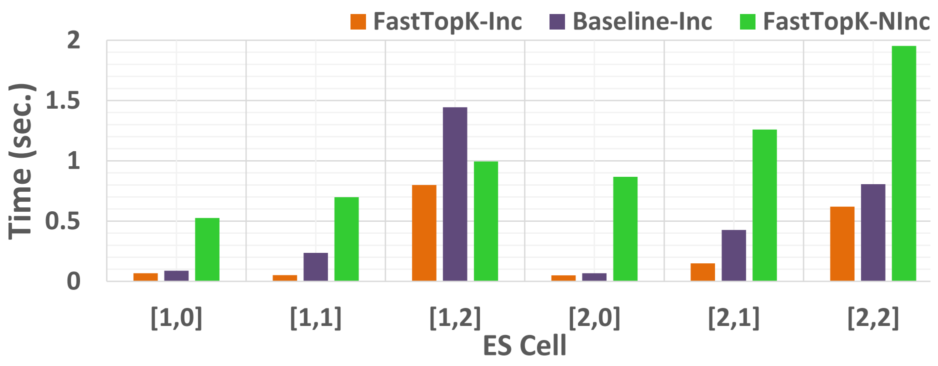


Figure 11.11: Execution time for incremental input [row, column]

We evaluate three approaches for incremental input:

(i) FASTTOPK-NINC: always treating an example spreadsheet as a new one and applying FASTTOPK on it;

(ii) FASTTOPK-INC: described in Section 11.5.4 and above; and

(iii) BASELINE-INC: extending BASELINE using the same ideas as FASTTOPK-INC without caching-evaluation scheduling

Refer to Section 11.6.1 for the settings of experiments. Figure 11.11 shows the execution times of the three algorithms for the 6 cell additions via row-wise typing. FASTTOPK-INC significantly outperforms both BASELINE-INC and FASTTOPK-NINC. FASTTOPK-NINC performs poorly since it does not use previously computed scores for the unchanged example tuples. This is especially true for the first few cells in a new row (e.g., [2,0], [2,1]). In these cases, the incremental approaches evaluate the PJ queries only for the changed example tuple (only a few terms) while FASTTOPK-NINC evaluates them for both changed and unchanged example tuples. BASELINE-INC suffers as it does not share results of sub-PJ queries as in the non-incremental case.

### 11.7.2 Generalizing Cell Similarity

We can extend cell similarity to perform IR-style relevance ranking as in [Sin01]. For example, we can define cell similarity to be higher if there is an exact match between an example tuple cell and output row cell (as opposed to the latter only containing some terms of the former). And we can incorporate terms weights (based on inverse document frequency and document length) in cell similarity.

We can also extend cell similarity to handle spelling errors, synonyms, and fuzzy matching. For example, to handle fuzzy matching, we can built the inverted indexes (both column-level and row-level) on n-grams as terms instead of words [MSLN00]. When processing an example spreadsheet, we split each cell into n-grams (instead of words) and retrieve the inverted indexes corresponding to those n-grams. For spelling errors, we can simply replace a term in the example spreadsheet with a list of similar terms (within certain edit distance), look up them in our inverted indexes, and take the union of posting lists. Synonyms can be handled similarly.

### 11.7.3 AND v.s. OR Semantics

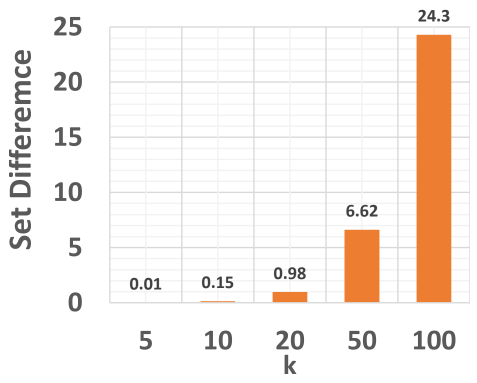
Our definition of PJ queries (Definition 4) requires that every column in the example spreadsheet is mapped to some column in the database. We can relax this constraint by allowing that only a subset of the example spreadsheet columns is mapped to the set of database columns, i.e., simply ignoring some example spreadsheet columns. Consider a PJ query $Q = (\mathcal{J}, \mathcal{C}, \varphi)$, this relaxation changes our mapping from "$\varphi : \mathrm{col}(T) \to \mathcal{C}$" to "$\varphi : \mathrm{col}(T) \to \mathcal{C} \cup \{\perp\}$". When a column $i$ in the example spreadsheet $T$ is mapped to $\perp$, this column does not correspond to any column in the output relation of $Q$. We call our old column mapping, *AND-column mapping*, and the new relaxed one, *OR-column mapping*.

Our approaches can be extended to handle OR-column mapping easily. A simple extension is as follows. For a user-given example spreadsheet $T$ with $c$ columns, our system can create $2^c$ example spreadsheets $T_0, T_1, \ldots, T_{2^c-1}$, each of which consists of a subset of columns of $T$. We then process them using our FASTTOPK strategy one by one. The $2^c$ top-$k$ resulting lists are aggregated to generate the overall top-$k$. Since $c$ is small in practice, the cost of this approach is affordable. A more direct way is to enumerate all PJ queries under this OR semantics. To this end, we can apply the *Candidate Network Generator* in

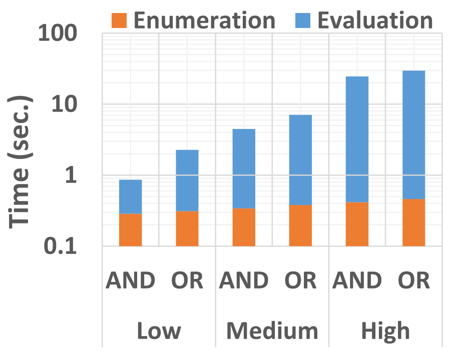
[HGP03] to generate an extended set of candidate PJ queries in $\mathcal{Q}_C{}^+$, each of which has a subset of example spreadsheet columns mapped to its projection. Then both of our strategies BASELINE and FASTTOPK still work on $\mathcal{Q}_C{}^+$.

## Experimental Results

Refer to Section 11.6.1 for the settings of experiments. We use CSUPP dataset and corresponding example spreadsheets to compare FASTTOPK with AND-column mapping (the one described in the main body of this chapter) with the extended version of FASTTOPK with OR-column mapping (the simple extension described above).



(a) Difference in result sets



(b) Execution time

Figure 11.12: AND-column mapping v.s. OR-column mapping

Figure 11.12(a) shows the average set difference between the result sets with AND and OR-column mapping for the 50 example spreadsheets for various values of $k$. For smaller values of $k$, there is almost no difference between the two result sets. For example, the top $10$ results are identical for 49 out of the 50 ESs. It means that, even when we allow OR-column mapping in PJ queries, the top ones are likely to have all the columns in example spreadsheets mapped to their projections (AND semantics).

Figure 11.12(b) shows the average execution times of the two approaches. Note that "enumeration" here means "query enumeration + upper-bound score computation". The OR-semantics implementation is only a bit slower than the AND-semantics one since the execution time for the $T_i$ created from the biggest subset of columns in $T$ (i.e., the entire example spreadsheet) dominates the execution time. This shows that our system can be easily adapted to support OR-semantics whenever necessary, e.g., when the system returns an empty result for some user-specified example spreadsheet.

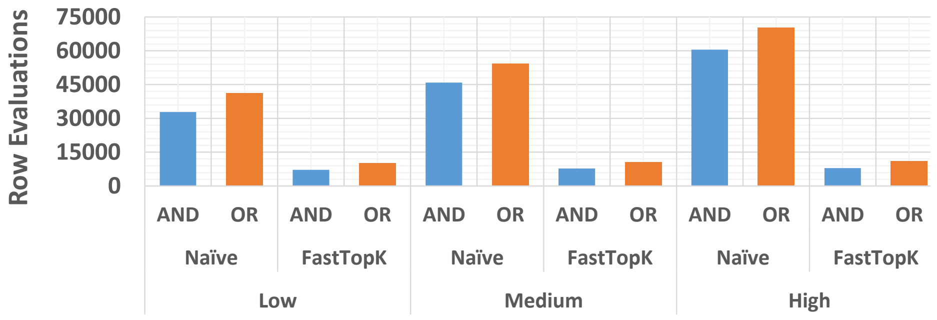


Figure 11.13: Amount of queries enumerated and evaluated in AND and OR semantics by NAIVE (not using $\overline{\mathsf{score}}$) and FASTTOPK (using $\overline{\mathsf{score}}$)

Figure 11.13 plots the number of PJ queries enumerated and evaluated in AND and OR semantics. NAIVE evaluates all the PJ queries enumerated in both semantics. We can find that, in the AND semantics, less PJ queries are enumerated than in OR because of stronger constraints in the column mapping $\varphi$. Using upper bounds $\overline{\mathsf{score}}$, the number of PJ queries actually evaluated by FASTTOPK is much less than the total number of queries enumerated.

## 11.8 Computing Exact Scores

### 11.8.1 Execution Plan for PJ Queries

We need to execute the PJ query $Q$ and, for each tuple $t$ in the example spreadsheet, examine every row in the output relation $\mathcal{A}(Q)$ to compute the terms $\max_{r \in \mathcal{A}(Q)} \mathsf{score}(t \mid r)$ in (11.3). We utilize the *(key, foreign key)-snapshot* of the database (discussed in Section 11.3.1), and select a pre-optimized plan to execute $Q$ in memory. Our execution plan for $Q$ borrows ideas from hash joins:

**Stage I** ***(scanning row-level inverted indexes to score cells)*****:** For each term $\mathrm{w}$ in each cell $\mathrm{t}[\mathrm{i}] \in \mathrm{T}$, suppose column $\mathrm{i}$ is mapped to column $\mathrm{j}$ in a relation $\mathrm{R}$ in the database $\mathcal{D}$ through $\varphi$, we retrieve the row-level inverted index $\mathsf{inv}(\mathsf{w}, \mathrm{R}[\mathrm{j}])$ to compute cell similarities $\mathsf{score}_{\mathrm{cell}}(\mathrm{t}[\mathrm{i}] \mid \mathrm{r}[\mathrm{j}])$ (as lines 1-5 of Algorithm 1) for rows $\mathrm{r} \in \mathrm{R}$. $\mathsf{score}_{\mathrm{cell}}$ is then associated to primary keys of rows in $\mathrm{R}$.

**Stage II** ***(bottom-up hash joins)*****:** Starting from the leaf relations in $\mathcal{J}$, the primary key of each row, associated with cell similarities, is inserted into a hash table if cell similarity is non-zero in at least one cell – after that, a leaf relation is called *evaluated*.

Recursively, **Stage II-A** ***(scan/hash lookup)*****:** for each relation above, if all of its children relations have been *evaluated*, we can start to *scan* its rows in the in-memory (key, foreign key)-snapshot of this relation, and for each row, *look up* all foreign keys (in different columns) in the corresponding hash tables popped up from the children relations to conduct the foreign-key joins. **Stage II-B** ***(building hash table)*****:** Then, the primary key of each row in the join output with nonzero cell similarities (in at least one cell) is put into a hash table. After that this relation is called *evaluated*.

**Stage III** ***(computing scores)*****:** After the root relation is evaluated, we get the output relation $\mathcal{A}(\mathrm{Q})$, with cell similarities associated in each row of $\mathcal{A}(\mathrm{Q})$, and then we can compute the row containment score as in (11.1)-(11.3). Note at at each relation, we only need to keep primary/foreign key columns and columns in the projection $\mathcal{C}$. □

The above evaluation plan can be executed either for $\mathrm{Q}$ with one row of $\mathrm{T}$, or with all rows of $\mathrm{T}$ together.

Figure 11.14 shows the execution plan for the PJ query in Figure 11.2(b)-(i) on the first row of the example table in Figure 11.2(a). A rectangle node represents the operation to retrieve row-level inverted indexes and compute cell similarities in Stage-I. A circle node represents the operations (scanning, hash lookups, and building hash table) we perform on a relation (labeled beside) in Stage-II, after all of its children are evaluated. For example, on the Orders node, we lookup foreign key Orders.CustId of each row in the hash table built by the node Custmer, and then build a hash table with Orders.OId (key in the hash table)

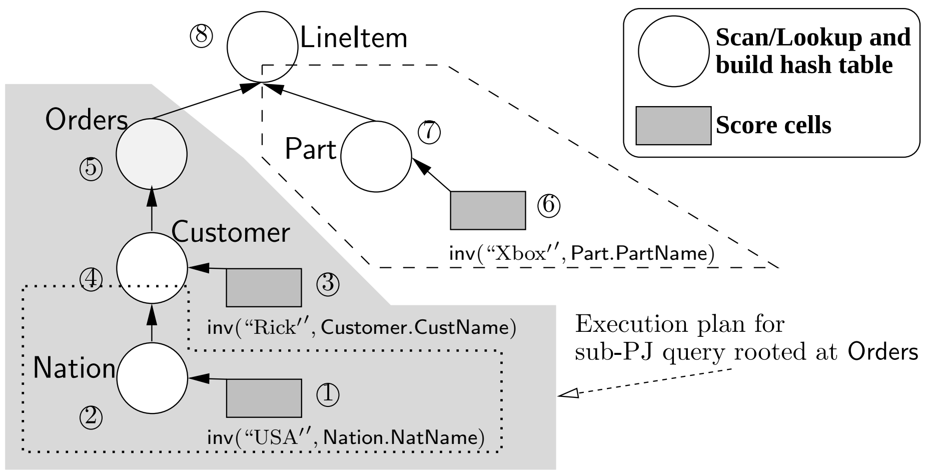

⑧ LineItem
Orders
⑤
Part
⑦
⑥
inv("Xbox", Part.PartName)
Customer
④
③
inv("Rick", Customer.CustName)
Nation
②
①
inv("USA", Nation.NatName)
Scan/Lookup and build hash table
Score cells
Execution plan for sub-PJ query rooted at Orders

Figure 11.14: Execution plan for the PJ query in Figure 11.2(b)-(i) and its sub-PJ queries (operators executed in the order of 1, 3, 6, 2, 4, 5, 7, 8)

and cell similarities on column Customer.CustName and column Nation.NatName. On the root node LineItem, we need to look up two foreign keys PartId and OId in the hash tables popped up by its two children Part and Orders, respectively. Operations are performed in the order of ①, ③, ⑥, ②, ④, ⑤, ⑦, ⑧ to compute the final score.

## 11.8.2 Speedup Execution using Cache

The execution plans of PJ-queries can be easily extended to take advantage the cached sub-PJ queries: for a PJ query $\mathrm{Q}$ and a set of cached sub-PJ queries in $\mathcal{M}$, instead of starting from the leaves of $\mathrm{Q}$, we start from the output relations of maximal sub-PJ queries of $\mathrm{Q}$ in $\mathcal{M}$ and follow the execution plan of $\mathrm{Q}$ afterwards.

For example, Figure 11.3 shows two sub-PJ queries $\mathrm{Q}'_1$ (left) and $\mathrm{Q}'_2$ (right) of the PJ query $\mathrm{Q}$ in Figure 11.2(b)-(i). Intuitively, the execution plan of a sub-PJ query $\mathrm{Q}' \preceq \mathrm{Q}$ is a subtree of the execution plan of $\mathrm{Q}$. For example, the dotted polygon and the dashed polygon in Figure 11.14 are the execution plans of $\mathrm{Q}'_1$ and $\mathrm{Q}'_2$, respectively.

In the execution plan of $\mathrm{Q}$ in Figure 11.14, if both $\mathrm{Q}'_1$ and $\mathrm{Q}'_2$ are materialized in $\mathcal{M}$, we can start from their output relations in $\mathcal{M}$ and execute only the operations ③, ④, ⑤, and ⑧ (only the join with Part is needed in ⑧). If a even larger one $Q'_3$ (rooted at Orders), whose plan is the shaded polygon in Figure 11.14, is cached, we can start from the output relations of $Q'_1$ and $Q'_3$ ($Q'_2$ is no more a maximal one) and execute only the operation ⑧.

## 11.8.3 Cost Model for Computing Exact Scores

For the PJ-query $Q = (\mathcal{J}, \mathcal{C}, \varphi)$ w.r.t. $T$, there are three major operators involved in our execution plan in 11.8.1: i) retrieving row-level inverted index; ii) scanning a relation while doing hash lookups; and iii) building a hash table. The running time of each of these operators is constant. So a natural and light-weight cost model of the execution plan for $Q$ is to count the number of operations ii) and iii) executed on tuples in the relations of $\mathcal{J}$ and the number of tuples retrieved from inverted indexes. For a (sub-)PJ query $Q$, define the cost of evaluating $Q$ as:

$$\mathrm{cost}(Q) = \sum_{R \in \mathcal{V}(\mathcal{J})} |R| \cdot d_{\mathcal{J}}(R) + \sum_{i \in \mathrm{col}(T)} \sum_{w \in T[i]} |\mathsf{inv}(w, \mathcal{J}[\varphi(i)])|. \tag{11.12}$$

The first component on the RHS of (11.12) quantifies the total number of hash lookups/inserts: $d_{\mathcal{J}}(R)$ is the degree of relation $R$ in $\mathcal{J}$, as for each tuple in $R$, the number of hash lookups is equal to the number of children of $R$ in $\mathcal{J}$, and for every relation except the root relation in $R$, we need to build a hash table by inserting tuples in this relation. The second component quantifies the number of tuples we need to retrieve from row-level inverted indexes: here let $\mathcal{J}[\varphi(i)]$ be the column which $i$ is mapped to in a relation of $\mathcal{J}$.

More generally, we have a set of sub-PJ queries cached in $\mathcal{M}$. The cost of the execution plan for $Q$ when we reuse output relations of sub-PJ queries in $\mathcal{M}$ is defined to be:

$$\mathrm{cost}(Q, \mathcal{M}) = \mathrm{cost}(Q) - \sum_{\substack{\text{maximal } Q' \in \mathcal{M} \\ Q' \preceq Q}} \mathrm{cost}(Q'), \tag{11.13}$$

as the output relations of maximal sub-PJ queries $Q'$ of $Q$ in $\mathcal{M}$ can be directly retrieved from $\mathcal{M}$ and reused.

Both $\mathrm{cost}(Q)$ and $\mathrm{cost}(Q, \mathcal{M})$ can be computed efficiently. In (11.12)-(11.13), $|R|$, the number of tuples in $R$, and $|\mathsf{inv}(\mathsf{w}, \mathcal{J}[\varphi(i)])|$, the length of a row-level inverted index, can be gotten in constant time. So the total time is $O(\mathcal{V}(\mathcal{J}) + \#\text{ terms in } T)$.

## 11.9 Proofs

**Proof of Proposition 1.** For the first part (property i)), since no column in $T$ is mapped to $R$, by excluding $R$ from $Q$, the column containment score is unchanged, i.e., $\mathsf{score}_{\mathrm{col}}(T \mid Q) = \mathsf{score}_{\mathrm{col}}(T \mid Q')$. So it suffices to prove $\mathsf{score}_{\mathrm{row}}(T \mid Q) \leq \mathsf{score}_{\mathrm{row}}(T \mid Q')$. Consdier the output relations $\mathcal{A}(Q)$ and $\mathcal{A}(Q')$, for any $t \in \mathcal{A}(Q)$, we have $t \in \mathcal{A}(Q')$, because $Q'$ has less key-foreign key constraints in joins. So from Equation (11.1), we have, for each $t \in T$, $\mathsf{score}(t \mid Q) \leq \mathsf{score}(t \mid Q')$. Then the conclusion follows from Equation (11.3)

For the second part (property ii)), $\mathsf{score}_{\mathrm{col}}(T \mid Q) = \mathsf{score}_{\mathrm{col}}(T' \mid Q'')$ is also obvious as no term in the removed column $T[i]$ appears in the removed $R[j]$. It suffices to prove $\mathsf{score}_{\mathrm{row}}(T \mid Q) = \mathsf{score}_{\mathrm{row}}(T' \mid Q'')$. Comparing $Q$ with $Q''$, their join trees are the same and the projection in $Q''$ is a subset of the projection in $Q$. So the output relation $\mathcal{A}(Q'')$ is essentially the projection of $\mathcal{A}(Q)$ on $\mathcal{C}'$. And since the column $R[j]$ excluded in $Q''$ contains no term in the spreadsheet column $T[i]$, the above claim follows. □

**Proof of Proposition 2.** From Equation (11.5), it suffices to show $\mathsf{score}_{\mathrm{row}}(T \mid Q) \leq \mathsf{score}_{\mathrm{col}}(T \mid Q)$. Putting Equation (11.2) into Equation (11.3), and comparing it with Equation (11.4), we can derive this relationship. □

**Proof of Proposition 3.** For each term $\mathsf{w}$ in a column $i$ of $T$, if $T[i]$ is mapped to $R[j]$, we need to scan the row-level inverted index $\mathsf{inv}(\mathsf{w}, R[j])$ – the term in Equation (11.4), $\mathsf{score}_{\mathrm{cell}}(t[i] \mid r[\varphi(i)])$, is obtained by aggregating the results for different terms. The total cost is dominated by $O(\sum_{\mathsf{w} \in T} l_{\mathsf{w}})$, i.e., lengths of inverted indexes. □

**Proof of Proposition 4.** Again, the row-level inverted indexes need to be scanned to compute the terms $\mathsf{score}_{\mathrm{cell}}(t[i] \mid r[\varphi(i)])$ in Equation (11.2). To compute the terms $\max_{r \in \mathcal{A}(Q)} \mathsf{score}(t \mid r)$ in Equation (11.3), we need to scan the output relation $\mathcal{A}(Q)$ at least once. The complexity of generating $\mathcal{A}(\mathrm{Q})$ using the hash-join execution plan introduced in Section 11.8.1 is $\mathrm{O}(\sum_{\mathrm{R}\in\mathcal{J}} |\mathrm{R}| \cdot \mathrm{d}_{\mathcal{J}}(\mathrm{R}))$. □

**Proof of Proposition 5.** Let's first define the class of algorithms, called *multi-step ranking* algorithms. A multi-step ranking algorithm takes i) a set of PJ queries $\mathcal{Q}_\mathrm{C}$, and ii) upper bounds $\overline{\mathsf{score}}$ of their scores, as input. In each step, it picks one or more PJ queries in $\mathcal{Q}_\mathrm{C}$ with unknown scores and *evaluates* them, i.e., computes $\mathsf{score}(\mathrm{Q})$; based on the known scores, it continues to pick the next one or more PJ queries to evaluate, until the *top*-$\mathrm{k}$ of known scores is larger than the *max* of upper-bound scores of queries with unknown scores. The following proof follows from [SK98].

Recall $\mathcal{Q}_\mathrm{min} = \{\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_{\mathrm{i}^*}\} \subseteq \mathcal{Q}_\mathrm{C}$, and $\mathrm{i}^*$ is the minimal $\mathrm{i}$ s.t. $\mathrm{top}_\mathrm{k}\{\mathsf{score}(\mathrm{Q}_1), \ldots, \mathsf{score}(\mathrm{Q}_\mathrm{i})\} > \overline{\mathsf{score}}(\mathrm{Q}_{\mathrm{i}+1}) \geq \overline{\mathsf{score}}(\mathrm{Q}_{\mathrm{i}+2}) \geq \ldots \geq \overline{\mathsf{score}}(\mathrm{Q}_\mathrm{N})$. We prove this proposition via contradiction. Consider any multi-step ranking algorithm that evalutes a set of PJ queries $\mathcal{Q}'$ and claims that all queries with the top-$\mathrm{k}$ scores in $\mathcal{Q}_\mathrm{C}$ have been found in $\mathcal{Q}'$. Pick any $\mathrm{Q}_\mathrm{p} \in \mathcal{Q}_\mathrm{min} - \mathcal{Q}'$. Let $\mathrm{Q}_\mathrm{q}$ be the PJ query in $\mathcal{Q}'$ with the kth highest score, i.e., $\mathsf{score}(\mathrm{Q}_\mathrm{q}) = \mathrm{top}_\mathrm{k}\{\mathsf{score}(\mathrm{Q}) \mid \mathrm{Q} \in \mathcal{Q}'\}$.

i) If $\overline{\mathsf{score}}(\mathrm{Q}_\mathrm{p}) \geq \mathsf{score}(\mathrm{Q}_\mathrm{q})$: Since the algorithm has not evaluated $\mathrm{Q}_\mathrm{p}$, the adversary can set $\mathsf{score}(\mathrm{Q}_\mathrm{p}) = \overline{\mathsf{score}}(\mathrm{Q}_\mathrm{p})$. Then $\mathsf{score}(\mathrm{Q}_\mathrm{p}) \geq \mathsf{score}(\mathrm{Q}_\mathrm{q}) = \mathrm{top}_\mathrm{k}\{\mathsf{score}(\mathrm{Q}) \mid \mathrm{Q} \in \mathcal{Q}'\}$, so $\mathrm{Q}_\mathrm{p}$ is missed from the top-$\mathrm{k}$ and the output of the algorithm is incorrect.

ii) If $\overline{\mathsf{score}}(\mathrm{Q}_\mathrm{p}) < \mathsf{score}(\mathrm{Q}_\mathrm{q})$: Consider the set of top-$\mathrm{k}$ PJ queries in $\mathrm{Q}'$, $\mathcal{Q}'_\mathrm{topk} = \{\mathrm{Q} \mid \mathsf{score}(\mathrm{Q}) \geq \mathsf{score}(\mathrm{Q}_\mathrm{q})\} \cap \mathcal{Q}'$. We have $\mathcal{Q}'_\mathrm{topk} \subseteq \{\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_{\mathrm{p}-1}\}$, because for any $\mathrm{Q} \in \mathcal{Q}'_\mathrm{topk}$, we have $\overline{\mathsf{score}}(\mathrm{Q}) > \overline{\mathsf{score}}(\mathrm{Q}_\mathrm{p})$. Then it follows that $\mathrm{top}_\mathrm{k}\{\mathsf{score}(\mathrm{Q}_1), \ldots, \mathsf{score}(\mathrm{Q}_{\mathrm{p}-1})\} > \overline{\mathsf{score}}(\mathrm{Q}_\mathrm{p})$. Note that $\mathrm{p} \leq \mathrm{i}^*$ which contradicts with the minimality of $\mathrm{i}^*$.

Both i) and ii) lead to contradiction, so we have $\mathcal{Q}_\mathrm{min} \subseteq \mathcal{Q}'$. □

**Proof of Theorem 2.** To prove the correctness, we only need to show that if Equation (11.7) is satisfied, the top-$\mathrm{k}$ are among $\mathrm{Q}_1, \mathrm{Q}_2, \ldots, \mathrm{Q}_\mathrm{i}$. This is true, because, from the way how $\mathrm{Q}_\mathrm{i}$'s are ordered in Equation (11.7) and BASELINE, for any $\mathrm{j} > \mathrm{i}$, we have $\mathsf{score}(\mathrm{Q}_\mathrm{j}) \leq \overline{\mathsf{score}}(\mathrm{Q}_{\mathrm{i}+1})$. The second part, of the theorem (i.e., BASELINE evaluates only queries in $\mathcal{Q}_\mathrm{min}$) is trivial. □

**Proof of Theorem 3.** Given a sequence of type-a,b,c operators, to check whether it is a feasible solution, we only need to check when it eventually evaluates all PJ queries in $\mathcal{Q}_{\mathrm{min}}$ and, at any time, the cache size it uses is no more than $\mathrm{B}$. So when $\mathcal{Q}_{\mathrm{min}}$ is known, the problem is in NP. We use a reduction from the HAMILTONIAN PATH problem to show CACHE-EVAL SCHEDULER is NP-hard.

Consider a HAMILTONIAN PATH instance: given a undirected graph $\mathrm{G(V, E)}$, whether there exist an ordering of all vertices $\mathrm{v_1, v_2, \ldots, v_n \in V}$ ($\mathrm{|V| = n}$) s.t. $\mathrm{(v_i, v_{i+1}) \in E}$ for $\mathrm{i = 1, \ldots, n-1}$. This problem is NP-complete even in a restricted class of graphs, where every vertex has degree equal to three [GJS74]. Now let's construct an instance of our CACHE-EVAL SCHEDULER problem. For each vertex $\mathrm{v \in V}$, create a PJ query $\mathrm{Q_v}$ in $\mathcal{Q}_{\mathrm{min}}$. For each edge $\mathrm{e = (v, u)}$, create a sub-PJ query $\mathrm{Q_e^{sub}}$ and let $\mathrm{Q_e^{sub}}$ be a sub-PJ query of both $\mathrm{Q_v}$ and $\mathrm{Q_u}$. So for each created PJ query $\mathrm{Q_v}$, we have the set of all sub-PJ queries of $\mathrm{Q_v}$ to be $\mathcal{T}(\mathrm{Q_v}) = \{\mathrm{Q_e^{sub}} \mid \mathrm{e \text{ is incident on } v}\}$. Finally, let $|\mathcal{A}(\mathrm{Q_e^{sub}})| = \mathrm{B}$ for every $\mathrm{e \in E}$, i.e., at any time, we can only keep the output relation of one sub-PJ query in our cache; and let $\mathrm{cost(Q_e^{sub}) = C_1}$ be equal for every $\mathrm{e}$ and $\mathrm{cost(Q_v) = C_2}$ be equal for every $\mathrm{v \in V}$. To prove the NP-hardness, it suffices to show such a claim: there is a Hamiltonian path in $\mathrm{G}$ if and only if there exists a sequence of operators to evaluate $\mathcal{Q}_{\mathrm{min}}$ with cost no more than $\mathrm{n \cdot C_2 - (n-1) \cdot C_1}$.

To prove the claim, we need to transform a path into a sequence of operators and vice versa. A subpath $\mathrm{v_{i-1}v_iv_{i+1}}$ in the Hamiltonian path corresponds to: when evaluting $\mathrm{Q_{v_{i-1}}}$, we put $\mathrm{Q^{sub}_{(v_{i-1},v_i)}}$ in cache and reuse it to evaluate $\mathrm{Q_{v_i}}$; and after that, we clear the cache and put $\mathrm{Q^{sub}_{(v_i,v_{i+1})}}$ in cache. Details are omitted here. □

**Proof of Theorem 4.** The correctness follows directly from Theorem 2. The set of PJ queries FASTTOPK evaluate is always a superset of those evaluated by BASELINE (i.e., all queries in $\mathcal{Q}_{\mathrm{min}}$ have been evaluated when FASTTOPK terminates). □

**Proof of Theorem 5.** The second part, $\mathrm{M_{all} = \Theta(s_{max} \cdot N)}$, is directly from our definition of sub-PJ queries.

For the overall time complexity, we focus on Algorithm 4 first. For each sub-PJ query $\mathrm{Q}'$, we need $\mathrm{O}(\mathrm{s}_{\max})$ time to get $\mathrm{cost}(\mathrm{Q}')$ and $|\mathcal{A}(\mathrm{Q}')|$, which are used in lines 1 and 4. So we need a total of $\mathrm{O}(\mathrm{M}\cdot\mathrm{s}_{\max})$ time for all sub-PJ queries. Sorting all sub-PJ queries in line 1 needs $\mathrm{O}(\mathrm{M}\cdot\log\mathrm{M})$ time. The remaining question is how to get $\mathrm{Q}^*$ and $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$ efficiently (lines 4, 6, and 8). Note that $\mathcal{B}_\mathrm{j}$ is the set of unevaluated PJ queries. For each sub-PJ query $\mathrm{Q}'$, we keep a hash set $\mathrm{HS}(\mathrm{Q}')$ of *unevaluated* PJ queries it belongs to. For each PJ query $\mathrm{Q}$, we also keep a list $\mathrm{LT}(\mathrm{Q})$ of sub-PJ queries it contains. So after a PJ-queriy $\mathrm{Q}$ is evaluated, each $\mathrm{HS}(\mathrm{Q}')$ can be updated in constant time if $\mathrm{Q}'$ is a sub-PJ query of $\mathrm{Q}$. To get $\mathrm{Q}^*$ in all iterations, we only need to scan all sub-PJ quries $\mathrm{Q}'_1,\ldots,\mathrm{Q}'_\mathrm{M}$ in order and check whether $|\mathrm{HS}(\mathrm{Q}')|>1$ for each. $\mathrm{Critical}^{-1}(\mathrm{Q}^*)$ can be directed retrived from $\mathrm{HS}(\mathrm{Q}^*)$. So the time complexity of Algorithm 4 is $\mathrm{O}(\mathrm{M}(\mathrm{s}_{\max}+\log\mathrm{M}))$.

In Algorithm 3, the additional time besides invoking Algorithm 4 is at most $\mathrm{O}(\mathrm{N}\cdot(\log\mathrm{N}+\log\mathrm{k}))$ (soring plus keeping the top-k).

So from $\mathrm{x}\log\mathrm{x}+\mathrm{y}\log\mathrm{y}\le(\mathrm{x}+\mathrm{y})\log(\mathrm{x}+\mathrm{y})$, we have the overall time complexity is $\mathrm{O}(\mathrm{N}+\mathrm{M}_{\mathrm{all}}(\mathrm{s}_{\max}+\log\mathrm{M}_{\mathrm{all}}))$. □

**Proof of Theorem 6.** The first part Equation (11.9) is directly from the definition of $\mathcal{Q}_{\min}$ and the way we construct batches in Algorithm 3.

To prove the main result Equation (11.10), we first prove, for each batch $\mathcal{B}$

$$\mathrm{cost}_{\mathrm{TOT}}(\mathcal{B})-\mathrm{cost}_{\mathrm{SOL}}(\mathcal{B})\ge\frac{1}{2\mathrm{c}^2}\left(\mathrm{cost}_{\mathrm{TOT}}(\mathcal{B})-\mathrm{cost}_{\mathrm{OPT}}(\mathcal{B})\right). \tag{11.14}$$

That is, the cost saved by FASTTOPK is no less than $1/2\mathrm{c}^2$ of the optimal save. Summing up Equation (11.14) for all batches $\mathcal{B}=\mathcal{B}_0,\mathcal{B}_1,\ldots$, since they are disjoint and $\cup\mathcal{B}_\mathrm{j}=\mathcal{Q}_\mathrm{E}$, we have

$$\begin{aligned}&\mathrm{cost}_{\mathrm{TOT}}(\mathcal{Q}_\mathrm{E})-\Sigma_{\mathcal{B}_\mathrm{j}}\mathrm{cost}_{\mathrm{SOL}}(\mathcal{B}_\mathrm{j})\\ \ge&\frac{1}{2\mathrm{c}^2}\left(\mathrm{cost}_{\mathrm{TOT}}(\mathcal{Q}_\mathrm{E})-\Sigma_{\mathcal{B}_\mathrm{j}}\mathrm{cost}_{\mathrm{OPT}}(\mathcal{B}_\mathrm{j})\right).\end{aligned} \tag{11.15}$$

FASTTOPK works batch-by-batch, so we have i) $\sum_{\mathcal{B}_\mathrm{j}}\mathrm{cost}_{\mathrm{SOL}}(\mathcal{B}_\mathrm{j})=\mathrm{cost}_{\mathrm{SOL}}(\mathcal{Q}_\mathrm{E})$. Since $\mathcal{B}_\mathrm{j}\subseteq\mathcal{Q}_\mathrm{E}$, we have ii) $\mathrm{cost}_{\mathrm{OPT}}(\mathcal{B}_\mathrm{j})\le\mathrm{cost}_{\mathrm{OPT}}(\mathcal{Q}_\mathrm{E})$. And it is obvious that iii) the total number of batches processed by Algorithm 3 is at most $\log_{1+\varepsilon}(|\mathcal{Q}_{\min}|/k)$. Our performance ratio Equation (11.10) follows from Equation (11.15) and i)-iii).

The only missing part is the proof for Equation (11.14). Let the batch of PJ queries $\mathcal{B} = \{Q_1, \ldots, Q_n\}$. Suppose in the optimal solution,

$$\mathrm{cost}_{\mathrm{TOT}}(\mathcal{B}) - \mathrm{cost}_{\mathrm{OPT}}(\mathcal{B}) \leq \sum_{Q_i \in B} \mathrm{cost}(Q_i) - \mathrm{cost}(Q_i, \mathcal{M}_i^*), \tag{11.16}$$

where $\mathcal{M}_i^*$ is the status of cache before $Q_i$ is evaluated. From Equation (11.13),

$$\mathrm{cost}(Q_i) - \mathrm{cost}(Q_i, \mathcal{M}_i^*) = \sum_{\substack{\text{maximal } Q' \in \mathcal{M}_i^* \\ Q' \preceq Q_i}} \mathrm{cost}(Q'). \tag{11.17}$$

On the other hand, in Algorithm 4, when $Q_i$ is to be evaluated, we have cache $\mathcal{M}_i = \{Q^*\}$ (lines 8-9), where $Q^*$ is the most costly sub-PJ query among those in $\{\text{maximal } Q' \in \mathcal{M}_i^* \mid Q' \preceq Q_i\}$ based on its selection (line 4). There are at most $c$ maximal sub-PJ queries for $Q_i$ (as there are at most $c$ leaves in the join tree), so

$$\mathrm{cost}(Q_i) - \mathrm{cost}(Q_i, \mathcal{M}_i) \geq \frac{1}{c}\left(\mathrm{cost}(Q_i) - \mathrm{cost}(Q_i, \mathcal{M}_i^*)\right). \tag{11.18}$$

Among all PJ queries in $\mathcal{B}$, in the worst case, a fraction $\frac{1}{2c}$ of them can benefit as much as Equation (11.18) from the cache. So putting Equation (11.18) back to Equation (11.16), we can obtain Equation (11.14). The proof can be completed. □

## 11.10 Conclusion And Future Work

In this chapter, we proposed and studied the problem of discovering top-$k$ project join queries which approximately contain a user-given set of example tuples in their outputs. The main technical challenge is to share results of common subexpressions among the PJ queries and still terminate early. We formalize the problem as the caching-evaluation scheduling problem, show its hardness, and develop a near-optimal solution. Our experiments demonstrate that our solution is both efficient and effecitive in finding the top-$k$ PJ queries. Our ranker captures some classes of errors; extending it to other types of errors (e.g., spelling errors and fuzzy matching) and evaluating its quality on real enterprise users are open challenges.

## 11.11 Retrospective Analysis

Having described S4 in detail, we now provide our retrospective analysis to answer how instrumentation-enabled engines, such as SMOKE, could have helped in its development.

As we showed experimentally, the main problem behind the performance of S4 lies on the computation of the exact scores and, more specifically, on the computation of the row score similarity for every project-join query wrt to a given spreadsheet. To compute this similarity, S4 needs to evaluate thousands of join queries and this computation needs to happen interactively when users search through the spreadsheet interface. Executing such queries against a database engine without our indexes takes considerable time, rendering query discovery a non-interactive process. To account for this problem, we built two indexes, namely, row-level inverted index and primary key-foreign key (pk-fk) snapshot of the database that overall allowed us to perform thousands of joins per second and overall compute the similarity scores of project-join queries interactively.

In fact, most of the codebase of S4 and most of the time spent on its development was on implementing the logic behind building these indexes and using them in the query plans of project-join queries to compute the overall scores. Regarding building these indexes, we had to crawl (using projection; group-by; and join queries) the underlying data warehouses, ship projection; group-by; and join results over to S4, and manipulate the results to create the indexes. Note that both types of indexes have graph-like structures (i.e., row-level indexes are inverted indexes from terms to row ids, and pk-fk snapshots are inverted indexes from row ids of primary keys to row ids of foreign keys) and our query evaluation strategies had to incorporate them for the score computations. While these tasks may seem straightforward in comparison to the overall algorithmic development of S4, core performance benefits come from these tasks and we had to implement them from scratch to make S4 interactive.

Now, recall our provenance capture techniques over group-by queries. The row-level inverted index is exactly the backward rid index over grouping on distinct terms of column elements. This can be expressed with the query `SELECT T.C FROM T GROUP BY T.C`. The result of this query with backward provenance capture in SMOKE is a backward index mapping each term in `T.C` to the row ids in `T` that it came from. Hence, to construct the index we need to perform this query for every table `T` and column `C`. Note that, for consolidation purposes, we can also combine all indexes together by grouping on terms. Finally, note that each cell of a column may have multiple terms (e.g., full text). To account for such cases we first need to split terms per cell and then feed them to the group-by query above. (We can do so with an `unnest` operation on the list of split terms.) The end result is exactly the row-level inverted index of S4, and we only had to issue some queries to our provenance-enabled SMOKE database to create it.

Furthermore, recall our provenance capture techniques over join queries. The pk-fk snapshot index of S4 is exactly the which-provenance backward indexes over all possible two-way joins involving all possible primary key-foreign key combinations as specified in the schema of the database. Essentially, we map each row id of a primary key to the row ids of the foreign key table that have the same primary key in the joined column. As such, the construction of the pk-fk snapshot, in our provenance-enabled SMOKE database, would involve issuing a set of simple join queries over which we would capture which-provenance.

Having represented these indexes as backward provenance, we can now use them to compute scores of project-join queries using backward trace statements. Essentially, we need to compute intersections and unions over backward rid lists similar to the ones we described for crossfiltering (Chapter 8) and details on demand (Chapter 7). Such intersections and unions are evaluated efficiently in SMOKE following well-known techniques [WLPS17].

Our overall analysis highlights the premise of instrumentation-enabled engines in allowing application developers to focus on tasks that are inherent to their goals (e.g., query discovery in this case) as opposed to spending time writing databases from scratch only to embed their logic within physical operators.

# Chapter 12

# Other Connections and the Road Ahead

We conclude our discussion over applications domains, by drawing the connections between instrumentation and the domains of negative provenance (Section 12.1), online query optimization (Section 12.2), and the general domain of interactive applications (Section 12.3). Across domains, we discuss how well-known techniques can be expressed in an instrumentation-based way—hence, further evaluating the expressiveness of our instrumentation framework and demonstrating best practices—and we introduce novel extensions and semantics that instrumentation-enabled engines could enable in a principled manner—hence, covering interesting future directions.

## 12.1 Negative Provenance

Negative provenance [CWH$^+$17; HCDN08; CJ09] is a fundamental type of information that allows applications to answer questions such as "why an input tuple has not contributed to outputs". Applications of negative provenance include network analytics [CWH$^+$17], data debugging [AHS12], causality [MGS11; MGMS10], and integrity repairs [XZAT18;

MGS11], to name a few. Unfortunately, without instrumentation mechanisms to allow capturing negative provenance information within plans, state-of-the-art negative provenance capture systems typically rewrite queries to their negated forms and track positive provenance by operating at the logical level. As we demonstrated in Chapter 3, capturing provenance at the logical level comes with performance penalties, however. Such performance penalties are even higher for negative provenance capture because negative provenance needs to be captured per intermediate operator (i.e., which records where filtered per intermediate operator). Hence, provenance capture needs to be applied on the negations of subplans.

Negating and tracking positive provenance per intermediate operator, however, is not necessary shall a database provides instrumentation points on data flows that are not generated by the query execution, as we introduced in SMOKE in Section 6.3.

```
SELECT   COUNT(*), states.name, states.polygon
FROM     ontime, airports, states
WHERE    flights.origin = airports.iata and
         intersects(Point(airports.x, airports.y),
                          states.polygon);
GROUP BY states.name, states.polygon;
```

Figure 12.1: Flights per state.

To further highlight the importance of negative provenance and illustrate how it can be captured through physical plan instrumentation, consider again our `flights` database and the query in Figure 12.1 that counts the delayed flights per state. To do so, it finds within which state each airport lies by intersecting the location of each airport with the polygon of each state. The problem with the above query is that polygons can have arbitrary precision on bounding a state (or region in general). As a result, if we have selected a resolution for polygons that leaves out airports, then the overall results and the insights we can get by visualizing a heatmap of the counts of flights per state may be erroneous.

In fact, this is the case shall we instantiate our database schema with delayed flights from the ontime [Ont] dataset, airports and airlines from OpenFlights [Ope], and state

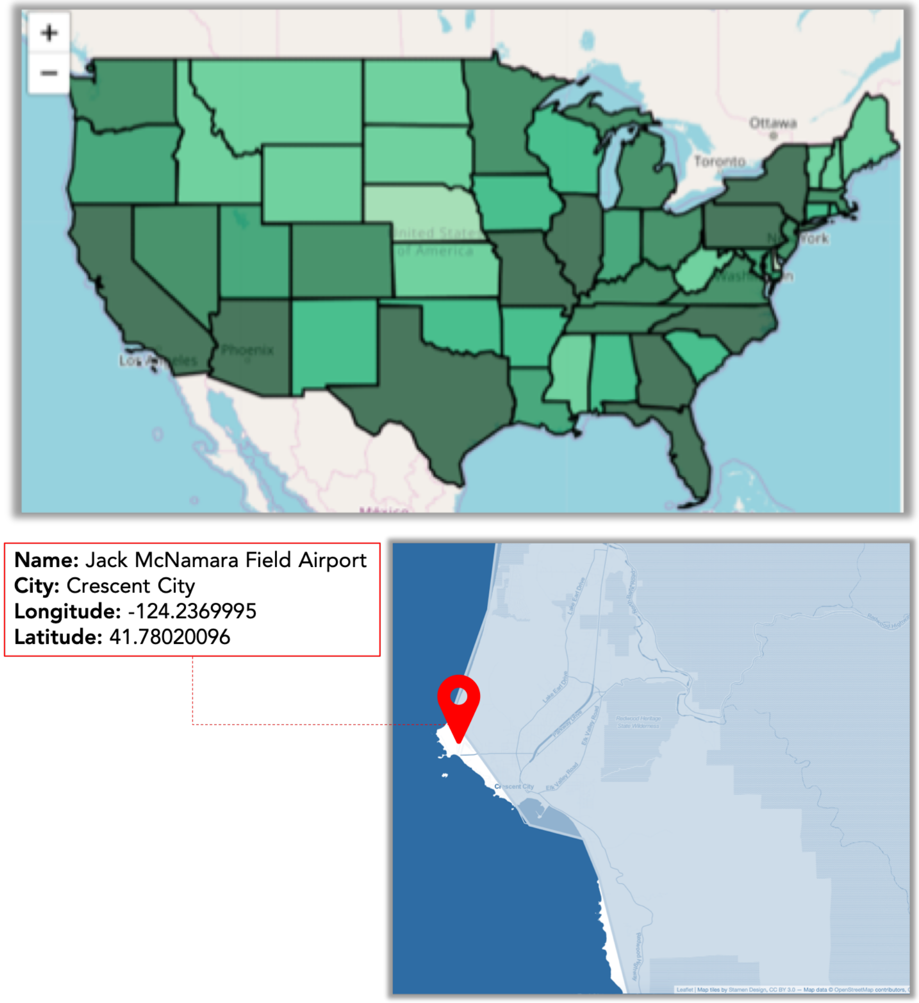
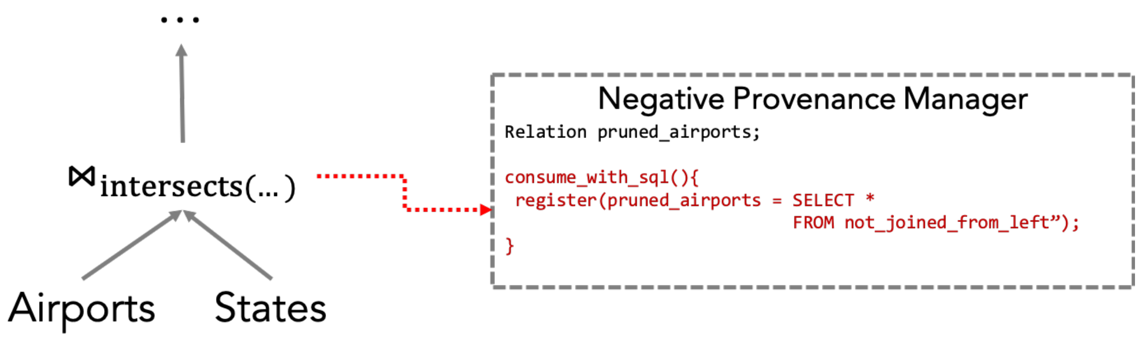


Figure 12.2: A heatmap of delayed flights for our example. Zooming in reveals that the Jack McNamara Field Aiport is not covered by the state polygons and not included in our results of delayed flights per state. As a result, insights extracted from the heatmap may be erroneous.

polygons from the US Census Tiger shapefiles [USC], which is the typical setting in many experimental studies. Consider the heatmap in Figure 12.2(top). At first sight, nothing seems to be wrong with the heatmap. However, if we zoom-in in Crescent City we can see that the Jack McNamara Field Airport is not included in our results. The visual explanation of this problem is depicted in Figure 12.2(bottom). The airport is not included in our results because the polygons from the US Census Tiger shapefiles have low resolution and the `intersects()` function of our query filters out the Jack McNamara Field Airport.



```
Relation pruned_airports;

consume_with_sql(){
 register(pruned_airports = SELECT *
                            FROM not_joined_from_left");
}
```

Figure 12.3: Capturing negative provenance for our example using instrumentation.

To provide this explanation to the user we need to track negative provenance at the moment we are executing our initial query. To express this in SMOKE, consider the plan in Figure 12.3(left) and the negative provenance manager in Figure 12.3(right). By using the `not_joined_from_left` instrumentation point of the nested loop join operator that performs the intersection, we can track which airports failed to pass the intersection. In SMOKE, we can express this functionality both declaratively and imperatively, as we discussed in Section 6.8.2; Figure 12.3(right) shows a declarative approach that stores the airports that where filtered out by the instersection in the relation `pruned_airports`.

A final note on this example is that the `not_joined_from_left` instrumentation point inherits the schema of the left side and this may be problematic for explanation purposes. For instance, the city of an airport in our example has no role in the query and a database may push a projection before the join to remove the city from further consideration in the plan. This is problematic because the `pruned_airports` relation for negative provenance capture still needs the details of the airport so that it can show them as explanation to the user, as shown in Figure 12.2(bottom). This functionality can also be expressed in SMOKE. More specifically, recall how the Actions component of the Physical Plan Instrumentation Framework of SMOKE allows us to change the schema of operators only to be captured by instrumentation applications, as we discussed in Section 6.8.1. Using this functionality, we can get more attributes from the airports table to populate the `pruned_airports` table and provide better explanations to users.

To conclude our discussion on negative provenance, we note that while our discussion above is limited on negative provenance capture on the left side of a nested loop joins, SMOKE also provides negative points for selections, `HAVING` clauses (implemented as selections), and both right and left sides of both nested loop and hash-based joins, as discussed in Section 6.3. Extending this functionality for other types of operators (e.g., anti-joins and set difference) is interesting future work. Furthermore, SMOKE also provides deferring negative provenance on operators. Putting these functionalities in practice per domain is also interesting future work.

## 12.2 Online Query Optimization

Traditional query optimizers that take as input a query and decide on a physical plan for its implementation can make arbitrarily erroneous decisions [LGM+15] due to the absence of exact statistics and knowledge of runtime conditions (e.g., change in a memory budget).

To address this problem, there is a vast literature of online query optimizers that observe the query at runtime and change it in response to updated statistics or runtime conditions. More specifically, online query optimization techniques collect knowledge about a query (e.g., CPU counts and memory consumption, updated selectivity and cardinality estimates, or even complete data structures such as bloom filters and hash tables) as well as observe run time events (e.g., change of memory budgets or CPU and machine availability). Then, based on this knowledge, they make decisions on how to change a physical plan at runtime. Smooth scans [BGIA+18] collect statistics during selections and change selection scans to index scans, and vice versa. Adaptive joins as introduced in commercial databases [SQL18; Ora17], change nested loop joins to hash joins, and vice versa, at runtime. Sideways [IT08] and lookahead information passing [PDZ+18] techniques collect information from one operator to pass it over and optimize other operations in a plan. Finally, probabilistic predicates [LCKC18; LKC18] change selections, which are applied after expensive machine learning operators, to probabilistic predicates before the machine learning operators.

All these example techniques require fine-grained control over the runtime of a physical plan (e.g., to change its control flow, manipulate internal state of operator, and observe and respond to runtime events). In the absence of such mechanisms, however, each technique has been implemented in an ad-hoc way by changing the internals of a single database server.

To this end, in this section, we revisit popular optimization techniques (i.e., probabilistic predicates, adaptive joins, and information passing) to express core tasks of them using our Physical Plan Instrumentation Framework. Our discussion aims to illustrates best instrumentation practices to enable the implementation of online optimization techniques without having to deal with the complexities of changing the underlying database engine.

### 12.2.1 Probabilistic Predicates

Consider the query and plan in Figure 12.4. The query processes a corpus of `videos`, extracts boxes per frame and camera (i.e., `PRODUCE` **`cameraId`**, **`frameID`**, **`vehBox`**), featurizes each box (i.e., $F_1, F_2$), classifies each box to the type and color of the vehicle it contains (i.e., $C_1, C_2$), and selects only the frames with boxes containing vehicles with type SUV and color red (i.e., **`WHERE type`** = **`SUV AND color`** = **`red`**).

In queries such as the above, the machine learning components (e.g., the vehicle detector **`VehDetector`**, the featurizers $F_1, F_2$, and classifiers $C_1, C_2$) typically dominate the query execution cost. However, note that the selections **`type`** = **`SUV AND color`** = **`red`** will only consider frames with vehicles of type SUV and color red. Hence, the machine learning components will spend a considerable amount of time processing frames that do not include vehicles of interest to the final result. In traditional query optimization this problem is typically solved by performing selection pushdown. However, in cases such as the above selections cannot be pushed down since the initial corpus is not annotated with types and colors of vehicles in frames.

```
SELECT cameraID, frameID,
       C1(F1(vehBox)) AS type,
       C2(F2(vehBox)) AS color
FROM  (PROCESS videos
       PRODUCE cameraID,frameID,vehBox
       USING VehDetector)
WHERE type = SUV AND color = red;
```

$\texttt{videos} \rightarrow \texttt{VehDetector} \rightarrow \mathrm{F}_1, \mathrm{F}_2 \rightarrow \mathrm{C}_1, \mathrm{C}_2 \rightarrow \sigma_{\mathrm{type=SUV} \wedge \mathrm{color=red}}$

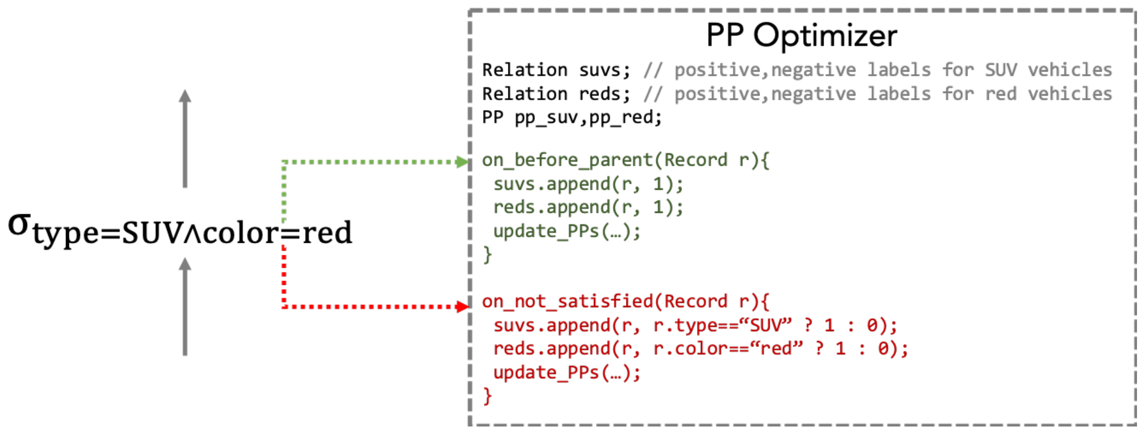
Figure 12.4: Example query (top) and corresponding plan (bottom) we use in our discussion. The query processes a corpus of videos to select the frames having vehicles with color red and type SUV. (Example borrowed from the original probabilistic predicates papers [LCKC18; LKC18].)

To address this problem, Lu et al. [LCKC18; LKC18] introduced the notion of probabilistic predicates (PPs). PPs is an online query optimization technique that, given plans involving such expensive machine learning components followed by selections on their final output, aims to push down the "selections" before the ML pipeline. Instead of pushing the selection, per the traditional selection pushdown optimization, they create and push their probabilistic alternatives. For instance, in the case above, we can build classifiers $\mathrm{PP}_{\mathrm{SUV}}$ and $\mathrm{PP}_{\mathrm{red}}$ that classify an input frame as containing vehicles of type SUV and color red, respectively. The premise is that such predicates can be cheap enough to construct and much cheaper than running the whole ML pipeline. The result is the plan shown in Figure 12.5.

$\texttt{videos} \rightarrow \mathrm{PP}_{\mathrm{SUV}}, \mathrm{PP}_{\mathrm{red}} \rightarrow \texttt{VehDetector} \rightarrow \mathrm{F}_1, \mathrm{F}_2 \rightarrow \mathrm{C}_1, \mathrm{C}_2 \rightarrow \sigma_{\mathrm{type=SUV} \wedge \mathrm{color=red}}$

Figure 12.5: The physical plan of our example after the introduction of probabilistic predicates $\mathrm{P}_{\mathrm{SUV}}, \mathrm{P}_{\mathrm{red}}$ before the expensive `VehDetector`.

Now, while there are many connections behind how our physical plan instrumentation framework can help in expressing the introduction of PPs in a plan (e.g., adding selections or changing their predicates online can be provided by the `add_operator` and CNF



```
Relation suvs; // positive,negative labels for SUV vehicles
Relation reds; // positive,negative labels for red vehicles
PP pp_suv,pp_red;

on_before_parent(Record r){
 suvs.append(r, 1);
 reds.append(r, 1);
 update_PPs(…);
}

on_not_satisfied(Record r){
 suvs.append(r, r.type==“SUV” ? 1 : 0);
 reds.append(r, r.color==“red” ? 1 : 0);
 update_PPs(…);
}
```

Figure 12.6: Getting positive and negative labels and updating PPs by instrumenting selections.

manipulation functions provided by the Actions component) here we will focus on how to construct PPs online to illustrate more complex features and best instrumentation practices.

The online construction of PPs lies on the fact that when we are first presented with a query, such as the one in Figure 12.4, we have no way of introducing PPs. Recall that PPs are classifiers. To build them we need some positive and negative labels first. For our example, we need to get the frames that pass the selection **WHERE** **type** = **SUV** **AND** **color** = **red**).

To perform this operation, recall the instrumentation points on selection Section 6.3.1 and our implementation of them in Figure 12.6. To get positive labels for either SUVs or red colored vehicles we can implement the `before_parent` or `after_parent` instrumentation functions of the selection **type** = **SUV** **AND** **color** = **red**. Every record that satisfies the selection provides positive labels. To get both positive and negative labels we can use the `not_satisfied` point on the selection. Every record that does not passes the selection can either have **type**!=**SUV** **AND** **color**=**red**, **type**=**SUV** **AND** **color**!=**red**, and **type**!=**SUV** **AND** **color**!=**red**. In Figure 12.6 we show a sketch of how this functionality can be implemented in an imperative way in SMOKE. By implementing the instrumentation functions `on_before_parent` and `on_after_parent` we can populate the relations `suvs, reds` that maintain positive and negative labels, respectively. Then, the techniques introduced in [LCKC18; LKC18] can use the relations `suvs` and `reds` to construct PPs.

By taking an instrumentation perspective on the construction of PPs, we can also see how different implementations of instrumentors can lead to novel techniques. For instance, instead of relying on classifiers that are built on the whole input we can use one-pass online classifiers to update PPs per record. As long as we have a guarantee on the accuracy of the current classifiers we can introduce PPs in the plan. Furthermore, note that PPs are predicated on the fact that machine learning pipelines are slow. However, to be sure about that monitoring of the machine learning operators is required since the machine learning operators are UDFs without a closed-form cost formula. Finally, as noted in [KEA+17] not every frame in a sequence of frames is equally important. By understanding the underlying sequence of frames or taking into consideration other underlying data statistics, which is logic that can be expressed within instrumentors, we believe there is ample space for future work on the optimization of PPs. Hence, instrumentation frameworks that facilitate their development and fast introduction in a database are important.

To conclude our discussion on probabilistic predicates, we note that the instrumentation-enabled techniques that we presented above allow PPs to be introduced in a database without having to change its internals. This is a very important result since the instrumentation framework that we introduced is not focused on just probabilistic predicates. This highlights the extensibility provided by an instrumentation-enabled database engine and the interesting future directions that we can take by (re)modelling tasks in an instrumentation-driven way.

### 12.2.2 Adaptive Joins

During query optimization, the optimizer may select a physical join implementation that is suboptimal for the query at hand. For instance, an optimizer may select a nested loop join in anticipation of low input cardinalities to the join operation in contrast to a hash-based join implementation because building hash tables may be expensive for the estimated input cardinalities. However, input cardinalities may have been underestimated substantially as a result of cardinality estimation errors in subplans [LGM+15]. To account for such errors, we can build an online optimizer that gets better estimates during runtime. Updated estimates, in turn, can result in identifying that the nested loop join has a higher cost than an equivalent hash join implementation. Based on this information, we want to replace the nested loop join with a hash-based one. This is the main idea behind adaptive joins [SQL18; Ora17] that change from one join implementation to another during query execution based on updated statistics tracked at runtime.

Similarly to the other query optimization techniques that we discussed above, adaptive joins, while important, they have been implemented within specific databases by changing their internals. Next, we discuss how adaptive joins can be implemented in an instrumentation-based way by discussing how hash joins can replace nested loop joins and how these decisions can be triggered by statistics collected at run time. Furthermore, we discuss how hash joins can change to nested loop joins, to target cases when memory budgets for hash tables used in hash joins are not enough. Changing, however, the underlying join implementation is only one way to optimize the join in this case. In this direction, we show how to introduce compression of hash tables to decrease the memory footprint of hash joins instead of changing them to nested loop ones.

### Nested Loop Joins (NLJ) $\rightarrow$ Hash Joins (HJ)

A naive approach to change an NLJ to a HJ one is by stopping the NLJ and rerunning the join with HJ. Essentially, this technique is similar to pre-flight replacement. There are two problems with this approach. First, parent operators of the NLJ may have already consumed partial results of the NLJ. In this case, we have to search all parent operators until a blocking parent operator, remove their state, reinitialize them, and only then execute the hash join. Of course, re-execution is not always possible because the first blocking operator may be the root node of the plan in which case results have been sent to client applications, and a database engine cannot have control on the state of client applications. Second, re-execution in the form of delete state and re-execute misses optimization opportunities as partial results have already been created and re-execution wastes time recreating them. For these reasons, we need to introduce techniques that guarantee the continuation of the execution of the HJ from where the NLJ left off, as we also discussed in Section 6.8.3.

Consider a scalar in-memory implementation of NLJ for a natural equi-join between the outer relation A and the inner relation B. The natural join scans A and, for each tuple in A, it scans B for matches. To do so, the NLJ maintains a state (i.e., the current left and right input tuples in the nested loops). Changing NLJ to HJ has two cases based on which relation we want to make the probe side and which the build side of the HJ.

**A probe, B build.** Assume a HJ implementation that builds the hash table on B and probes it with tuples of A. To guarantee continuation, the HJ needs to build the hash table on the B side, as usual, but it needs to probe the hash table only with the tuples of A that the NLJ has not considered in its outer loop. To do so, we first can either introduce a new scan on B and feed it on the hash table construction or use the current scan of B. Both approaches are possible. The only problem with the second approach is that we have to reinitialize the scan to rescan from the start. Once the hash table is constructed, we can ask the current scan on A, that was previously used to feed NLJ, to continue producing tuples which we will now use to probe the hash table of HJ.

**A build, B probe.** Furthermore, assume an HJ implementation that has A on the build side and B on the probe side. In this case, we don't need to build the hash table on all of A. Rather, we only need to build only on the A tuples that have not yet been considered by the NLJ. To do so, we can continue the scan on A from where NLJ left it off. Then, we can restart the scan on B and probe the hash table on A to perform the join. Note however a case that needs extra consideration here. The last inner loop on B for the A tuple, say, $\mathrm{a}_{\mathrm{last}}$ that the NLJ left off may have not finished yet. In this case, the technique we discussed above may output duplicate join results involving $\mathrm{a}_{\mathrm{last}}$ and B tuples for which the last inner loop has already passed over. To address this issue, we simply first finish the inner loop for $\mathrm{a}_{\mathrm{last}}$ before proceeding with the change to HJ.

Besides the change between join implementations, we also need to base our decision on statistics tracked during query execution. For our example, one type of such statistics is an update on the join cardinality estimate. Given an update of such estimate, we would like to set up a condition (e.g., the re-estimated join selectivity deviates a lot from the estimated join selectivity rendering the remaining cost of the NLJ to be higher than the HJ one).

To perform these operations (i.e., change between join implementations, track statistics, and setting up conditions), we need to operate directly on the physical plan. This is where the instrumentation framework that we introduced in Chapter 6 comes into place.

To track the join cardinality, we can instrument the NLJ operator to get the number of tuples that pass and do not pass the join predicate. We have already shown how to perform this operation in Chapter 6. To set up a condition on the deviation of the re-estimated join cardinality from the initially estimated cardinality and get notified when it is met, we can use the `on` function of the Announcer component in Section 6.7. Recall that the `on` function takes as input a `Condition`, a `ResolveFunction`, and parameters to be passed to the `ResolveFunction` when the `Condition` is met. For our example, the `ResolveFunction` function of the Announcer has to implement the logic that we described above for replacing NLJ with HJ.

```
on(join_cardinality > thr, my_replace, {NL,A,B});
void my_replace(PhysicalOpPNode NL,A,B){
  nl_parent = NL.parent; // keep the parent
  remove_operator(NL, SINGLE); // remove NL
  add_operator(B, ⋈ht); // add hash table building as parent of B
  add_operator(A, ⋈probe); // add probe as parent of A
  ⋈ht.parent = ⋈probe; // add probe as parent of build
  ⋈probe.parent = nl_parent;  // the parent of probe is the NL parent
}
```

Figure 12.7: Changing NLJ to HJ using the physical plan instrumentation framework of SMOKE.

We show this specification in Figure 12.7. The `on` function takes as input the condition `join_cardinality > thr`, where `thr` expresses the deviation from the initially estimated join cardinality. Note that more complicated conditions are beyond our discussion. Here, we only want to show how such conditions can be expressed in SMOKE. Furthermore, the `on` function takes as input the function `my_replace` which will change the NLJ to

HJ once the condition is met. In Figure 12.7, we show a sketch of how to implement `my_replace` to change NLJ to HJ if we were to make the A relation the probe size and the B relation the build side, using operations from the Actions component of SMOKE.

Finally, recall our discussion on equivalent under replacement operators in Section 6.8.3. SMOKE exposes replacements, such as NLJ to HJ, directly and instrumentation applications do not need to provide their own replacement functions. For instance, in our example one could set the `on` function to use the replace of SMOKE instead of `my_replace`.

### Hash Joins (HJ) $\rightarrow$ Nested Loop Joins (NLJ)

Replacing a hash join with a nested loop join implementation is also important. More specifically, in real time systems where memory budgets can change at any given time, hash-based algorithms that use hash tables may need to swap to alternatives that use no extra memory for their computation. Next, we discuss changing HJ to NLJ when the memory footprint used by HJ has surpassed a given budget.

Changing HJ to NLJ has two cases. Changing HJ to NLJ if the build side of HJ has not finished yet is straightforward. We just replace NLJ to HJ, as we would do in a pre-flight replacement. Changing HJ to NLJ while HJ is probing has two cases based on what side will become the outer relation and which will become the inner relation of NLJ, similarly to the NLJ$\rightarrow$HJ replacement. In this case, however, both can be treated in the same way.

Assume that at the moment we want to change HJ to NLJ, the scan for probing on the probe side has stopped on the tuple $\mathrm{B_j}$. Based on $\mathrm{B_j}$, we have the following bi-partition of B tuples $\mathrm{B_C} = \{\mathrm{B_0}, \ldots, \mathrm{B_j}\}$ and $\mathrm{B'_C} = \{\mathrm{B_{j+1}}, \ldots, \mathrm{B_{|B|}}\}$, where $\mathrm{B_C}$ and $\mathrm{B'_C}$ denote the partitions that we have and not have probed with, respectively. What we need to do now is set the NLJ to account only for A and $\mathrm{B'_C}$; whether we put A as outer and $\mathrm{B'_C}$ as inner, or vice versa, is not affecting the correctness of the join. Expressing this change in SMOKE is similar to how we expressed NLJ$\rightarrow$HJ, and we omit further discussion.

Finally, to express the condition (i.e., the hash table memory is above a budget) recall that our Storage Manager allows accessing the state of operators to get statistics (e.g., size) of data structures maintained by SMOKE. As such, we can specify an `on` function again with the condition to be that the memory of a hash table is above a threshold.

**Other Actions and Hash Table Compression**

As a final note, the techniques that we discussed above only change one operator to another to account for knowledge acquired during the plan execution. However, such actions are only way to act upon knowledge acquired at runtime. In this direction, we note that the pattern illustrated by the `on` function (i.e., whenever a condition is met, perform an action) can account for the introduction of arbitrarily complex actions.

As an example, consider again that we use an HJ to perform a join but, at runtime, the space required by HJ exceeds a memory budget). In our discussion above, we handled this case by changing HJ to NLJ. However, we could also simply compress the hash table to drop its memory footprint below the memory budget. To do so, the `on` function should take a condition on the memory of the hash table, as above, but the `ResolveFunction` should be a hash table compressor. (Here, we only focus on expressing this functionality, and we omit a discussion on hash table compression.) Now, note that compressing a hash table means that parent operators of the HJ that still need to consume the join results in a non-compressed form are in jeopardy. In this direction, the `ResolveFunction` should not only compress the hash table but also add other operators (i.e., using the Actions component of SMOKE) to perform the decompression before the parents consume join results.

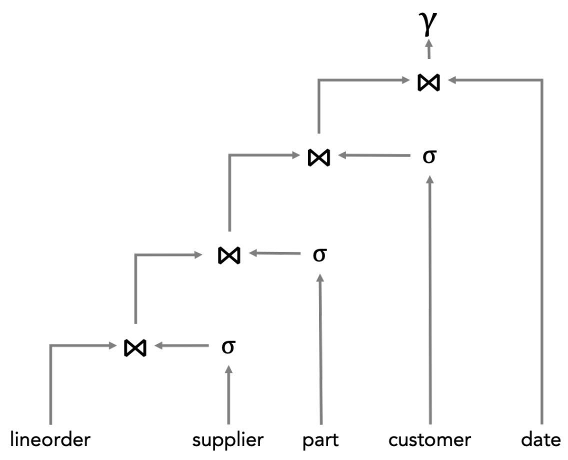
## 12.2.3 Information Passing

Complex query plans may involve operators out of which we can extract information that can be used for the optimization of other operations within the plan. This is the main idea behind information passing techniques, such as LOOKAHEAD [ZPSP17; PDZ$^+$18] and SIDEWAYS [IT08] information passing. While such techniques are powerful for the optimization of query plans they are only supported by specific databases, as we discussed in Chapter 1. In this direction, next, we show how to use the instrumentation capabilities of

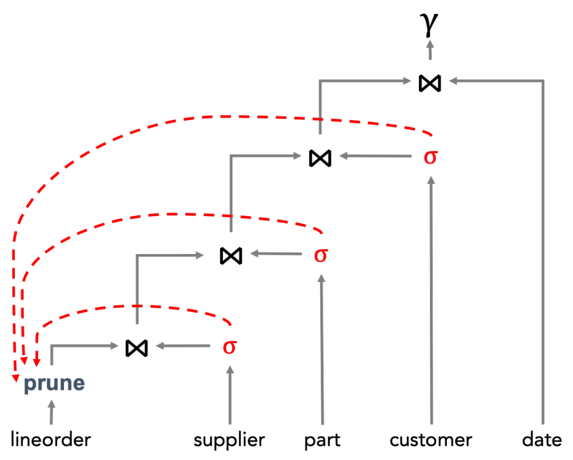
SMOKE to implement such techniques within our framework and without having to deal with the internals of the underlying database. To avoid redundancy on instrumentation practices in this space, we limit our discussion on lookahead information passing (LIP).

```
SELECT      d_year, c_nation,
            sum(lo_revenue) - sum(lo_supplycost) as profit
FROM        lineorder
LEFT JOIN   dates    ON lo_orderdate = d_datekey
LEFT JOIN   customer ON lo_custkey = c_custkey
LEFT JOIN   supplier ON lo_suppkey = s_suppkey
LEFT JOIN   part     ON lo_partkey = p_partkey
WHERE       c_region = 'AMERICA' AND
            s_region = 'AMERICA' AND
            (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
GROUP BY    d_year, c_nation;
```

(a) SQL specification



(b) Logical plan.



(c) Logical Plan with LIP optimization.

Figure 12.8: SSB query Q4.1: (a) SQL specification, (b) logical plan without LIP optimization, (c) logical plan with LIP optimization. Example borrowed from [PDZ+18].

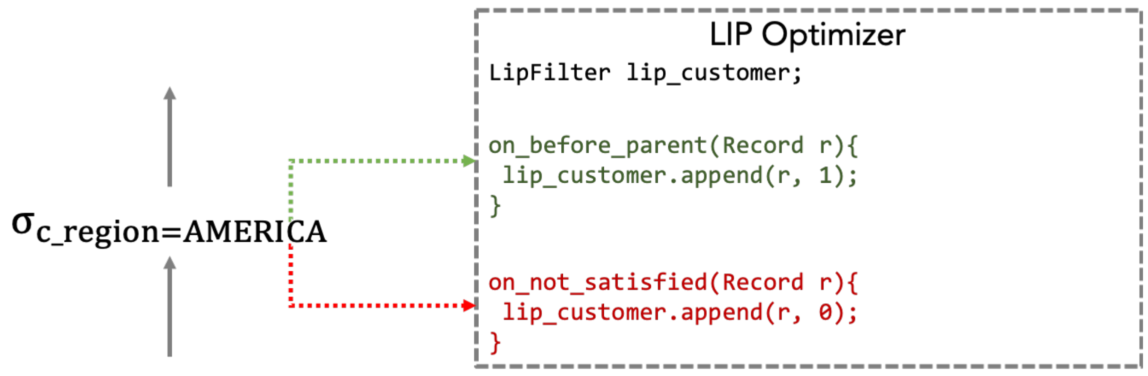
Consider the Query 4.1 from the star schema benchmark (SSB) [OOCR09], as illustrated in Figure 12.8a. The query joins the table `linorder` with the tables `supplier`, `customer`, and `dates` after it has applied selections on the tables `supplier` and `customer`. (Also, the query is followed by a grouping on `d_year,c_nation` and an aggregation to compute the profit per group. For our discussion on LIP the group-by aggregation is irrelevant; we use it here to highlight that optimizations can focus on subparts of a plan.)

The logical plan for the query Q4.1 is a left-deep plan, as illustrated in Figure 12.8b. A physical plan for this query first builds hash tables on the right sides (i.e., on $\sigma($`supplier`$)$, $\sigma($`customer`$)$, and `dates`) and probes these hash tables with records from the `linorder` table to evaluate the joins. (Note that Figure 12.8 shows only the logical plan for ease of presentation and compliance with the presentation of the LIP technique [PDZ+18]. We have omitted the corresponding and more complicated physical plan.)

Now, a core observation on this plan is that records from the `lineorder` that will be pruned out by, say, the join with the $\sigma($`customer`$)$ will pass earlier operations in the plan. As a result, a lot of space and time will be consumed early in the plan for records that have no effect on the overall result due to late pruning. For instance, for the `lineorder` records that will be pruned by the join with $\sigma($`customer`$)$ we will have to pay the costs of performing their join (e.g., probing the underlying hash table) with $\sigma($`supplier`$)$.

To address this problem, Zhu et al. [ZPSP17] introduced LIP techniques that encapsulate the core principle of “drop early, drop fast” (i.e., drop records early as opposed to waiting to be pruned late by plan operators). For our example, the LIP technique is shown in Figure 12.8c. The main idea is to construct LIP filters (i.e., bloom filters) during the selections and construction of hash tables from the right sides. Using these LIP fliters we can check which `lineorder` records will be pruned out by later joins. As such, we can push these filters down to the `lineorder` to avoid probing hash tables with `lineorder` records that will not be joined. This is illustrated with the **`prune`** operator in Figure 12.8c that takes as input the LIP filters from the right sides (illustrated with red arrows in Figure 12.8c).

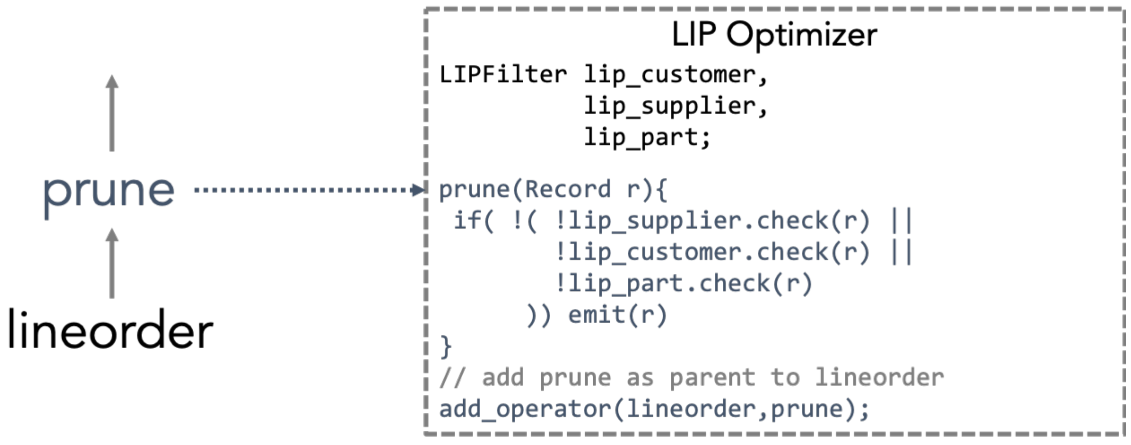
Now, let us consider how such techniques can be implemented in an instrumentation-based way. First, we need to construct the LIP filters (i.e., bloom filters) during the selection from the right sides. We illustrate how this can be done during the selection on `customer` in Figure 12.9. The main idea is to use the instrumentation points of the selection that we introduced in Section 6.3 and implement their corresponding instrumentation functions



```
LipFilter lip_customer;

on_before_parent(Record r){
 lip_customer.append(r, 1);
}

on_not_satisfied(Record r){
 lip_customer.append(r, 0);
}
```

Figure 12.9: Sketch for the construction of an LIP filter (i.e., bloom filter) during the selection $\sigma_{\texttt{c\_region=AMERICA}}$ using instrumentation points of the selection operator.

to construct the corresponding LIP filter. We illustrate a sketch of this implementation in Figure 12.9. Using the `on_before_parent` instrumentation function we can access the records that satisfy the selection and append the information to the LIP filter. Using the `on_not_satisfied` we can access records that did not satisfy the selection and append this information to the LIP filter accordingly. Similarly, we can also construct LIP filters for the selection on suppliers and parts.



```
LIPFilter lip_customer,
          lip_supplier,
          lip_part;

prune(Record r){
 if( !( !lip_supplier.check(r) ||
        !lip_customer.check(r) ||
        !lip_part.check(r)
      )) emit(r)
}
// add prune as parent to lineorder
add_operator(lineorder,prune);
```

Figure 12.10: Sketch for the implementation of the `prune` operator and its addition to the plan.

Now, we still have to introduce the prune operator in our plan. We can do so by implementing the operator in our LIP optimizer and adding the operator within the plan using the Actions component, that we introduced in Section 6.8. We illustrate the underlying implementation in Figure 12.10.

To conclude our discussion on information passing, we showed how to express an illustrative LIP technique without having to change the database internals using only the mechanisms provided by our instrumentation framework. By taking an instrumentation perspective on this domain, we believe there is ample space for future work such as information passing and inter-communication across different plans within the same or across different instrumentation-enabled database engines. As such, our instrumentation framework is a first step towards a potentially rich space of future optimizations.

## 12.3 Interactive Applications

In Chapter 7, we described how core visualization interactions can be succinctly expressed in provenance and instrumentation terms. This means that a visualization engine that is engineered to support provenance querying can readily add support for such interactions. Developers can then declaratively specify interactive visualizations and rely for their optimization on the underlying instrumentation- and provenance-enabled visualization engine. In this section, we look beyond existing interactive visualization features to examine *new functionality that may be possible* with the capabilities of such an engine. Finally, note that while most of our focus throughout this dissertation was on fine-grained provenance, coarse-grained provenance can also be derived from instrumentation (i..e., by extracting the descriptions from physical operators, as we discussed in Chapter 6). Hence, in our discussion next we will consider both forms of provenance in presenting novel interactions.

### 12.3.1 Advanced Provenance Analysis

To begin, we first highlight a rich area of provenance analysis techniques, such as interactive query specification [AHS12]; what-if analysis [AKLT15; DIMT13]; and result explanation [WM13] among others, that already exists. These techniques are a natural fit with

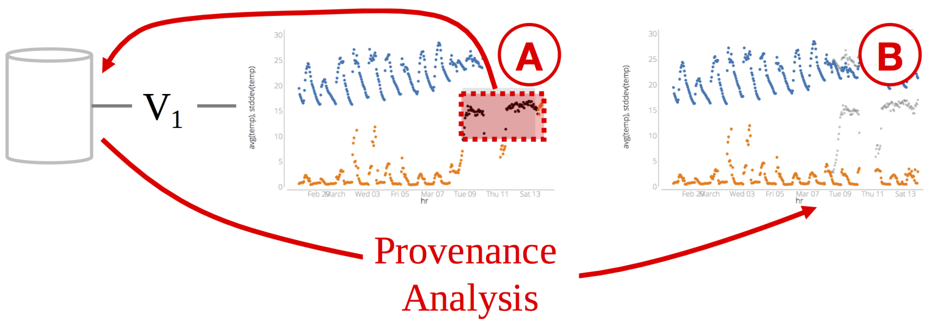


Figure 12.11: Before and after of an advanced provenance analysis. (a) the user selects outliers in the initial visualization (shown on the left), and (b) the results of the predicate explanation update the visualization (shown on the right). In practice, the visualization will update in place.

a provenance-enabled visualization engine (Figure 12.11). First, their inputs consist of provenance metadata and user-provided information that can be naturally elicited through a visualization interface. Second, their outputs are often in the form of predicates, records, or queries that can be naturally rendered in a visualization. Furthermore, they can be integrated as a function over the provenance result in a similar way to linked brushing and crossfiltering in Chapters 7 and 8. We illustrate a few examples of such integration below.

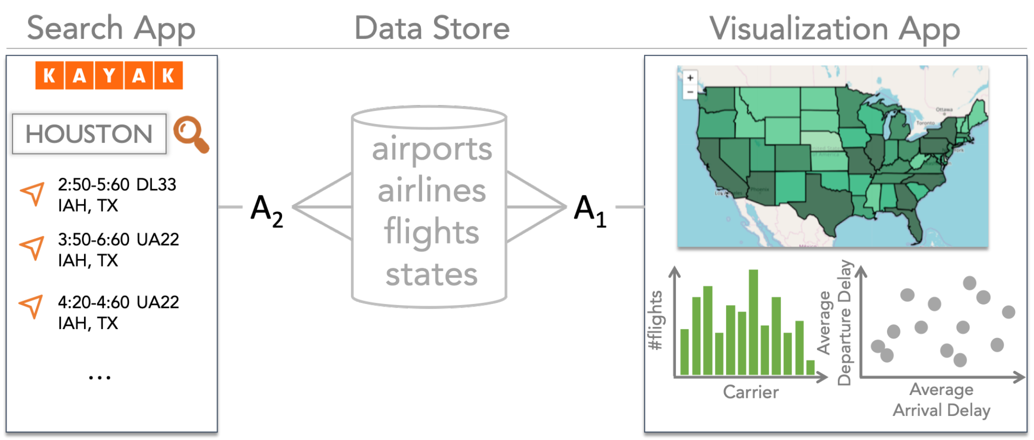
**Data Explanation.** Outlier explanation techniques [WM13; ROS15; WMS12] take as input anomalies in the visualized data, the query used to generate the visualization, and return simple predicates that are most "responsible" for those errors. Figure 12.11 shows how this is integrated into an interactive visualization. The user selects anomalies in the scatter plot on the left (A). Then, the data explanation analysis procedure uses $V_1$ and the fine-grained provenance of the selected points to generate a predicate explanation. Rather than rendering the explanation in textual form, we can also deeply integrated the explanation into the visualization itself. The example visualization in Figure 12.11 recomputes the query $V_1$ over a subset of the input identified by the explanation and renders it as an overlay (B).

**Why-not Analysis.** Non-existence of anticipated query results play a detrimental role in the overall data exploration and analysis. For instance, in Section 12.1, we showed the impact of an airport that was not contributing to the number of flights per state because the intersection with low-resolution polygons that we used to represent states geographically was pruning it out. Furthermore, if the state of California was missing in the map plot of Figure 12.2, then the user may be confused. Similarly, if the user complains that the `COUNT` of delayed flights should be higher for a specific carrier, perhaps by resizing its corresponding bar to be higher, then the user is questioning the absence of delayed flights in the visualization. Although the algorithms for generating these explanations may differ [AHS12; MGMS10], the way they can be integrated into, and presented within, the visualization are similar to the preceding example on data explanation. For instance, in Figure 12.2 in Section 12.1 we can click on a state on the map to check which airports within a state have not contributed to the result, use the negative provenance to identify the airports that are not contributing to the result, and finally present airport details with tooltips all while zooming in to focus the visualization on the location of identified airports, as shown in Figure 12.2(bottom).

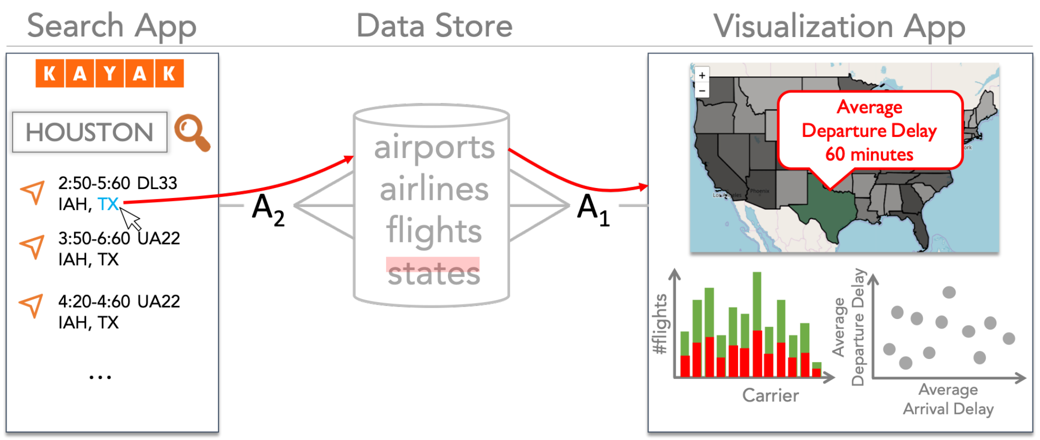
### 12.3.2 Multi-application Linking

Visualizations contain multiple views in order to present patterns between important combinations of attributes as we showed with linked brushing and crossfiltering interaction in Chapters 7 and 8. Such interactions are powerful because they assist users in identifying relationships between patterns, such as correlation statistics or data dependencies, visually.

In terms of functionality, they combine record-level backward tracing from selections in the visualization with forward tracing to (and refreshing of) visualizations dependent on shared input data. By expressing these forms interactions in terms of provenance it becomes clear that the backward and forward tracing as well as refresh operations need not be coupled, *nor even be implemented within the same visualization application*. As long as different applications process the same dataset, and the underlying database engine supports instrumentation and provenance functionality, then linking and crossfiltering *across multiple applications* is possible. This is particularly important for enterprise and academic settings with different applications getting developed over the same datasets.



(a) Different applications can be implemented on top of the same database.



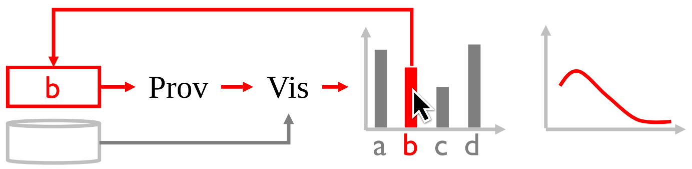
(b) Interacting in one application can result in updates on other applications.

Figure 12.12: Provenance can enable linking and crossfiltering across different applications.

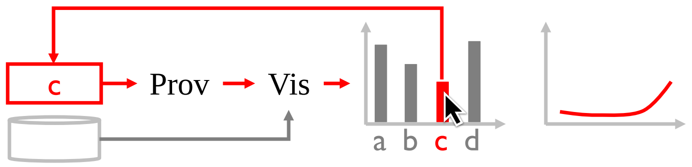
Figure 12.12 illustrates linking between the visualization application that we built in Chapter 7 over delayed flights with an external search application that allows users to search online for flights. (For our example, assume for now that the database contains both already performed and available flights.) The user may use a form-based search interface, as shown in Figure 12.12 (left), to find recent flights through Houston. This result set is fundamentally the result of a query workflow over the data store but presented as a text- and image-based web application. By tracing these search results back to the input data (the red rectangle over `states` represents the traced back subset of the relation), they can also be traced forward to update the visualization application (depicted by the red arrows). Furthermore, changing the search parameters may also update both the search results and the visualization. The reverse is also possible: selecting data in the visualization can also update the search results.

While our example focuses on an external search application, similarly any other application can be connected to our visualization and search applications. For instance, user profile tools may show to users their past flights and bookings can be linked to update the visualization to show delay statistics of the user's past flights, as well as to update the search results with flights the user has taken. Generalizing, any application that tracks backward provenance can issue interactions that update the presentation in any other application that supports forward provenance, as long as the two coordinate on the same base relations.

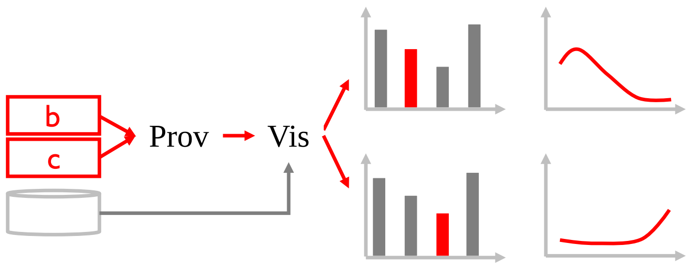
Finally, note that multi-application linking can also be supported when two applications are built on top of different databases as long as there are provenance connections between the databases. For instance, consider again our example in Figure 12.12. As we noted above, the database contains both already perform flights and flights available for booking. In the common setup, however, flights available for booking will be hosted in a different OLTP database. Then, when flights are performed and their delays are known they will be transferred to an OLAP database, so that we can build analytical tools (e.g., our visualization applications) and better understand underlying patterns. (For an overview of how applications are built within enterprises and how data are transferred from one type of database to another refer to [CDN11]). To support multi-application linking we need to track provenance during the transfer from the OLTP database to the OLAP database. Then, we can backward trace from the search application to the OLTP database, then back to the OLAP database, and finally forward to the visualization application. While what we described above is often regarded as a *pipe dream*, we believe that given the ever-increasing importance of provenance and its adoption across systems [Wid05; BCTV04; GA09; NKG+17; GKM06; Cui01; GKIT07; LDY13; IST+15; WMS13; IPW11; IW10; Ike12; WS97; CLMR16] such functionalities can now become a reality.



(a) Hovering over bar b triggers an interaction event to trace b's provenance (Prov) and update the line chart (Vis).



(b) Hovering over bar c performs the same logic but for the event associated with c.



(c) Explicitly tracking the provenance as a relation of events can easily render a history of past events.

Figure 12.13: Provenance of a crossfilter interaction can be modeled as the history of the visualization's interaction events.

### 12.3.3 Provenance of Interactions

So far, we have described how provenance can be *used* to express the results of interactions. For example, in our crossfiltering application in Figure 7.13 in Chapter 7 we showed that the bottom bar chart is updated by re-running $V_1$ over the backward provenance of the highlighted bars in the top bar chart. In many cases, interactions simply change the inputs to the application logic (e.g., $V_1$, $V_2$) rather than the logic itself. In these cases, interactions are a form of input data, whose provenance and versions can be tracked, By doing so we can change the logic of the application by visualizing the results of past interactions.

Figure 12.13 illustrates this for a simple crossfiltering visualization, where hovering over bars in the bar chart updates the line chart. We have simplified the workflow for clarity. `Vis` describes all application logic to compute and render both views; it is analogous to the union of $V_1$ and $V_2$ in Figure 7.13 in Chapter 7 for crossfiltering. When the user hovers over the `b` bar, the crossfilter logic executes $\mathrm{Vis}(\mathrm{Prov}(\texttt{b}))$ to update the visualization, shown as the red arrows in Figure 12.13a. The crossfilter logic is typically written within an event-handler that executes for each interaction event.[1] Thus, when the user hovers over bar `c`, the crossfilter logic simply executes $\mathrm{Vis}(\mathrm{Prov}(\texttt{c}))$, shown in Figure 12.13b.

```
SELECT Vis(Prov(e)) FROM events e
WHERE  e.source = 'barchart';
```

Figure 12.14: Query pseudocode to render history of interactions generated from the bar chart.

Now, note that the interaction events `b` and `c` are data, thus we might track the provenance of the visualization interactions in e.g., a relation of `events` (Figure 12.13c shows a relation containing `b,c`). This relation lets us decouple visualization update logic from user interactions, and manage them explicitly. For instance, Figure 12.13c shows how a history of past events can be presented and Figure 12.14 shows how it can be implemented in a relational manner. Similarly, selecting a single record is akin to undo or time-travel. Generalizing on this pattern, advanced functionality may also select a 2D-range of marks, and query for historical *interactions* (backward provenance to the `events` relation) that generated charts based on the selection (forward provenance to historical visualizations).

[1] In a relational context, where the visualization is modeled as a materialized view, as we discussed in Chapter 8 this is similar to scheduling view updates in response to changes in input relations.

### 12.3.4 Application Design Search

Tracking [HSG+17] and recovering [MCACM17; HKN+16] coarse-grained provenance in order to understand how workflows and applications throughout an organization result in reads and writes of data files. This can be helpful if a developer wants to analyze a given dataset, by suggesting previous workflows that have processed the same files. Similar functionality can help provide inspiration for visualization and application developers. For example, visualization developers that want to analyze flight delays for the North American marketing team can use coarse-grained provenance to find visualizations that use the flight relations. They can use these visualizations, such as Figure 7.1, to interactively specify the subset of the flight relations they want to work with. Based on this subset of records, fine-grained provenance can be used to identify the visualizations that primarily uses this specific subset. This iterative form of refinement can help the developer find the most relevant designs and application logic to borrow from, or perhaps find that their desired visualization already exists.

### 12.3.5 Interaction-By-Example

View synthesis and query-by-example systems [PDCC15; MLVP14; DFG17] address the problem where, given an input database and examples of desired query results, the goal is to return queries that generate the example results (or a superset). This formulation can be attractive because SQL queries are known to be hard to compose. This is due to the expressiveness and overall compositionality of SQL, and approaches typically focus on a semantically meaningful subset of the language for which identifying the queries by output examples can be efficient. For instance, as we showed in Chapter 11, our focus with S4 was to discover project join (PJ) queries because within the context of analytical databases with large schemas the main problem is to discover salient connections between data elements across many different relations. While the task is computational hard in the general case, we showed experimentally that our techniques can suggest top-$k$ PJ queries in interactive time for common and semantically meaningful cases.

Now, our overall discussion over interactive visualizations and applications has shown that we can express a wide range of visualization interactions in a blend of provenance and relational operations. Connecting these results with query discovery illustrates the potential to develop *interaction-by-example* as an interactive visualization discovery paradigm. For instance, we can allow users to directly select and manipulate parts of a static visualization (e.g., drag marks to new locations) to specify an example of a desired interaction, and underlying interaction discovery systems can suggest possible interactions composed out of provenance and relational operators. This is akin to [SVK$^+$07] but specific to fine-grained provenance rather than coarse-grained provenance. Such a synthesis engine can then generate the appropriate provenance statements blended with SQL to support the interaction. The simplicity of provenance queries suggests that this may be both tractable and semantically meaningful, and we expect future work in this direction with the goal set to address the challenging problem of composing interactive visualization applications.

### 12.3.6 Deconstruction and Restyling

We conclude our discussion over possible novel interactions, by presenting ways that provenance and instrumentation can help in deconstructing and restyling interactive visualizations. Harper et al. [HA14] present a technique to extract data from marks in D3 visualizations and re-style the data using new visual encodings. For instance, a bar chart might be restyled into a scatterplot that is colored differently. Their technique relied on D3 because it automatically annotates each mark with the record used to generate the mark. However, D3 does not track annotations across data processing workflows, thus restyling is limited to design. In this direction, we showed how to express interactive visualizations as relational workflows and track fine-grained provenance over them, which encodes the overall annotations of marks with input records. Furthermore, instrumentation can let users restyle a visualization based on changing the data processing parts. For example, we can plot `MAX` rather that `COUNT` statistics in our crossfiltering application, or modifying the semantics of linked interactions. This is possible through adding, removing, and modifying operators that SMOKE provides through its Actions component, as we discussed in Section 6.8.

## 12.4 Conclusions

In this chapter, we showed connections between instrumentation with the domains of negative provenance, query optimization, and interactive applications. More specifically, across domains we showed, how well-known and novel techniques can be introduced in a principle way within a database without requiring developers to change the database internal but, instead, using the instrumentation mechanisms of SMOKE. Furthermore, across domains and individual techniques, we highlighted best instrumentation practices as well as suggested future directions for expressing both novel and well-techniques. As such, our discussion suggests a first step towards the instrumentation-based optimization and introduction of both well-known and novel techniques for a large number of important application domains that rely their logic on how queries are executed by a database.

# Chapter 13

# Related work

In this chapter, we cover related work on instrumentation, provenance, physical database design, online query optimization, and data visualization. More specifically, we start by presenting related work in the areas of instrumentation in software development (Section 13.1) and in databases (Section 13.2) and discuss the main differences from our work. Then, we present related work in provenance (Section 13.3) and discuss the importance of instrumentation-enabled database engines in this domain. Furthermore, we cover related work on the domains of interactive data visualization (Section 13.4), physical database design (Section 13.5), online query optimization (Section 13.6), and interactive data profiling (Section 13.7) to present their current state and discuss how taking an instrumentation perspective in these domains can assist expressibility and optimizations.

## 13.1 Instrumentation in Software Development

Instrumentation is a powerful concept that allows third-party code to alter, analyze, and extend the functionality provided by a program, application, or system. Technically, instrumentation can happen on various underlying representations of a program lifecycle including source code instrumentation, instrumentation of representations used during compiling and linking, and binary instrumentation. As such, several instrumentation frameworks have been proposed, such as Intel Pin [LCM$^+$05], ATOM [SE94], Valgrind [NS03]; DynamicRio [Bru04], Etch [RVL$^+$97], LLVM instrumentors [TZW$^+$17; LA04], with the goals set to facilitate the development, management, and optimization of the vast amount of programs that require instrumentation capabilities. Such programs include profilers [GKM82], cache simulators [MC17; JCLJ08], debuggers [Sta86; LLV], memory checkers [NS03], address sanitizers [SBPV12], trace analyzers [Int], and dynamic re-compilers [BGA03], to name a few.

Using such instrumentation frameworks for query instrumentation, while possible, requires developers to instrument at the level of source code and binaries. As we argued in Chapter 2, source code and binary representations of queries are low level, making operations (e.g., changing a nested loop join to a hash join) harder to implement and optimize compared to instrumenting at the level of physical plans. In other terms, similarly to how compilers like LLVM and GDB provide different instrumentation mechanisms for each of their underlying IRs to deal with the different semantics exposed by each IR, so needs to be the case for databases shall we treat them as DSL compilers of SQL queries to physical plans. Every IR (e.g., SQL queries in textual form, ASTs, logical plans, physical plans, and even source code for query-compiled engines) needs to be coupled with IR-specific instrumentation mechanisms to best facilitate the development of techniques that need access to the semantics of each IR.

## 13.2 Instrumentation in Databases

Instrumentation is not a novel concept in databases either. For instance, PostgreSQL [Pos13] provides hooks where queries, ASTs, logical plans, and physical plans are redirected to third-party applications for instrumentation. Similar is the case for other data processing systems such as MonetDB [Mal18; Mon15b], MySQL [Mys18a; Mys18b], or Spark [Spa18]. The main difference from other instrumentation-enabled database systems is that SMOKE focuses on exposing mechanisms to applications to operate on IRs rather than simply sending IRs to applications to operate on them. Furthermore, as we showed in Chapter 6, the mechanisms that we exposed required changing underlying database components that instrumentation applications cannot perform by simply manipulating a given IR. Additionally, note that in this dissertation we focused on mechanisms over physical plans because our focus is on applications that rely their logic on how queries are executed. This is not to say that instrumentation mechanisms at other IR levels is not important. To the contrary, we believe there is ample space for research towards instrumentation mechanisms on different IRs (e..g., logical plans or all the way down to binaries) to account for other applications domains (e.g., mechanisms for instrumenting logical plans can be used to help applications that perform logical query rewriting while instrumentation on binaries can be used for security purposes).

## 13.3 Provenance

Provenance is a fundamental type of information that describes the connection between input and output data items of a computation. Large bodies of work have studied theoretical foundations of provenance [GKT07; Ike12; BKT01; GT17; CWW00; CCT09], data models for provenance representations [CWW00; MCF$^+$11; ABS$^+$06; Wid05], as well as systems and techniques for provenance capture and querying [Wid05; BCTV04; GA09; NKG$^+$17; GKM06; Cui01; GKIT07; LDY13; IST$^+$15; WMS13; IPW11; IW10; Ike12; WS97].

The importance of provenance is most exaggerated by its real-world applications. Virtually, any application that depends its logic on the connections between input and output data elements can be expressed in provenance terms. As such, provenance is (or can be) integral across applications including debugging [WMS13; KIT10; IST$^+$15; LDY13; CTV05; ZSF17; KBY17], data integration analysis [CWW00], diagnostics [TBEO$^+$17], auditing [EU 18], security [CWH$^+$17; KIT10], explaining query results [WM13; WMS12; ROS15; DFG17], data cleaning [CIOP14; HKW$^+$15], iterative analytics [CLMR16], and interactive data profiling [PBF$^+$15], among others as we showed. This ubiquity highlights the importance of instrumentation-(and by extension provenance)-enabled systems.

At their core, provenance systems need to capture the relationships between input and output data items of a workflow with the goal to streamline future queries over provenance. State-of-the-art approaches involve materializing the provenance graph (i.e., the connections between input and output) eagerly during workflow execution or lazily during provenance querying. In Chapters 3 to 5, we formally defined the problems of provenance capture and analysis, classified state-of-the-art techniques, and discussed their shortcomings. Here, we note that current provenance systems either incur high overhead to capture provenance, or provenance analysis costs, or both. These overheads are enough that application developers resort to hand-tuned implementations to hard-code provenance logic in their applications.

The main criticism behind introducing provenance capabilities within physical operators and in general systems (i.e., to follow the physical provenance capture approach) is that it requires significant changes in the underlying codebase [HDBL17]. In this direction, we showed that having an instrumentation-enabled database engine allowed us to introduce provenance operators in the database without changing its internals. The end result is a provenance-enabled database engine that is performant enough to provide orders of magnitude faster provenance capture and analysis over lazy and eager logical provenance capture alternatives. Furthermore, we showed experimental evidence that the performance of our provenance-enabled engine meets or even improves on the performance of hand-tuned implementations of data-intensive tasks across various domains that could have been expressed declaratively in provenance terms and optimized as such.

## 13.4 Interactive Data Visualization

Data visualization studies are primarily concerned with the transformation of raw data to visual representations (e.g., barcharts, scatterplots, heatmaps, and dendrograms) and how users can interact with the visual representations to gain fast insights. Traditional visualization toolkits (e.g., d3 [BOH11] or Qt [qt]) provide general purpose programming APIs that application developers can use to implement transformations of raw data to visual representations. These toolkits are typically coupled with native or separate event handling libraries (e.g., jQuery [jQu] or React [rea]) so that users can specify interaction logic to form interactive visualizations. Unfortunately, these approaches require imperative implementation of the visualization and interaction logic which leads to applications that are hard to implement, maintain, optimize, extend, and reason about [WHW16; WPM$^{+}$17]. Data visualization systems (e.g., Tableau [tab] or Excel PowerBI [Pow18]) aim to address the problem of ease of implementation by allowing users to load or connect to their datasets in order to visualize and interact with them using predefined visualizations and interactions. Unfortunately, they offer domain-specific interaction and visualization logic all while depend on underlying query processing engines that are not optimized for data-intensive interactive applications. More recent data visualization systems, such as Vega-lite [SMWH17] or tidyverse [Tid], allow for declarative specifications of visualizations and interactions. Although these systems offer considerable degrees of freedom for the specification of interactive visualizations they are limited by the expressive and optimization power of their underlying data processing capabilities.

In this direction, in Chapter 7, we introduced our declarative approach towards the specification of interactive visualizations. More specifically, we started by specifying interactive visualizations in SQL-like terms. While expressive enough to support interaction classes in a well-known taxonomy of interactions [YKSJ07], we discussed why specifying several classes of important, data-intensive interactions (i.e., interactive selections, multi-view linking, and logic over selections) in pure SQL-terms leads to interactions that are hard to express and optimize. To address this problem, we expressed these classes in a blend of SQL, provenance, and instrumentation terms and showed their performance benefits in Chapters 5, 8 and 10. Hence, an instrumentation-enabled engine (and by extension a provenance-enabled engine) is expressive enough to support the specification of interactions all while allowing their optimization through instrumentation capabilities. We believe our results illustrate interesting connections that the visualization systems discussed above [tab; SMWH17; Tid; BOH11] can exploit to further their expressive and optimization power.

## 13.5 Physical Database Design

The physical database design literature has long studied the problem of creating redundant data structures and data layouts either offline [CN07; ACN00; DRS$^+$05; Ora03; MGY15], online [BC07; CN07], or adaptively [KM05; IKM07a; SJLM16; LSJM17; AIA14; PIM15; VAPGZ17] with the goal to minimize the expected execution cost of a possible future query workload subject to constraints, such as space and time, on constructing redundant designs.

### Offline Physical Database Design

Offline physical database design is concerned with the construction of data structures (e.g., B-trees, hash tables, dictionaries, data cubes, and inverted lists) and layouts (e.g., through partitioning, chunking, and compression) given a future query workload (fixed or paramaterized). This is important for database applications with fixed or parameterized future query logic because it enables the construction of designs that can support any future query online. To address this problem, several offline physical database designers [CN07; ACN00; DRS$^+$05; Ora03; MGY15] have been proposed in the past that output a set of data structures and layouts that, if created, will optimize the performance of the future query workload. The decision of the design is subject to several constraints with the most important being the budget on the space occupied by the design. In the general case, this makes the decision an NP-hard problem with several techniques trying to approximate the decision either by formulating and addressing the problem as a knapsack variant [CN07]) or, more recently, through robust optimization and reinforcement learning [MGY15; SSD18].

While the central optimization problem is not tightly connected with what we have been discussing throughout this dissertation, we note that our provenance-based and instrumentation-based techniques result in physical database designs that offline physical database designers can use in their decision (e.g., data cubes, inverted indexes, join indexes, and denormalized representations). This highlights the extensibility of instrumentation-enabled engines towards the physical database design space.

Besides the construction of physical database designs, however, we also noted that offline physical database design depends on assumptions (e.g., infinite offline time, available DBAs to apply the suggestions, and that applications have future logic known a-priori [IPC15]) that may not hold in practice—especially in the space of interactive applications. This observation motivates the notion of online physical database design that we discuss next.

## Online Physical Database Design

Online physical design approaches [BC07; CN07] do not assume a fixed workload. Instead, they continuously monitor changes in the workload to update the physical design.

As we have already discussed, such techniques need monitoring capabilities (e.g., to extract runtime statistics such as CPU consumption or memory pressure), the ability to piggyback computations in physical plans for "execution feedback" (e.g., to gain insights over the underlying data distributions such as cardinalities [SLMK01] and histograms [AC99; BCG01]), and tradeoff the overhead of monitoring and gaining execution feedback with the query execution. Based on these mechanisms, several components are constructed (e.g., profilers [Pro], feedback caches [SLMK01; BCK$^+$11], and query progress estimators [CNR04; KDCN11]) that overall handle the online physical database design logic.

The connection between instrumentation-enabled engines and online physical database design is straightforward. Instrumentation engines provide the underlying mechanisms for the synthesis of the components that drive the online physical database design logic. More specifically, the interactive data profiling and provenance pushdown techniques that we discussed in Chapters 5 and 9 enable piggybacking of statistics. Furthermore, the components of our instrumentation framework allow for the implementation of more complicated monitoring, piggybacking, scheduling, and runtime management techniques. More precisely, Points and Instrumentors provide a principle way to implement more complicated monitoring and piggybacking techniques, the Scheduler allows deferring partially or fully the implementation logic to avoid the query execution overhead, and the Announcer can notify physical database designers with runtime events (e.g., for query progress estimation).

Despite these connections, online physical database design approaches generate the same physical designs as offline ones. Such designs, however, may take considerable time to construct, even if optimized by instrumentation engines, that upstream applications—especially interactive applications ones—may not tolerate. This observation motivates the space of adaptive physical database design that we discuss next.

## Adaptive Physical Database Design

Finally, adaptive approaches treat the current (or a batch of previous) queries as an indication of future queries similarly to online approaches. In contrast to offline and online approaches, adaptive approaches typically reuse prior results and data structures [GCZ$^{+}$17; DBCK17] or *incrementally* restructure the physical design [KM05; IKM07a; SJLM16; LSJM17; AIA14; PIM15] to avoid the excessive costs of creating in full data structures. Each adaptive design is fundamentally structured to support decisions on a limited set of current queries (e.g., database cracking techniques [SDL18; SJD13; PPI$^{+}$14; HIKY12; IKM07a; IKM07b; IMKG11; KM05] make design decisions based on selections) and target a limited set of future queries (e.g., selections on subsets of previous range selections).

While adaptive physical database techniques are important—especially for the domain of interactive applications—, there is little research besides cracking variants. In this direction, we showed that provenance capture is in effect a form of adaptive physical database design. More specifically, we showed how to generate physical database designs in the form of provenance indexes (Chapter 3), annotations (Chapters 3 and 4), or other physical database designs (e.g., rollup cubes) induced by our workload-aware optimizations (Chapter 5). Moreover, we showed how to perform adaptive physical database design over group-by aggregations to account for crossfilter interactions and increment cube exploration (Chapter 8), over group-by followed by distinct operators to account for evaluation and exploration of functional dependency and uniqueness profiling tasks (Chapter 9), over selections by introducing cracking frameworks to account for the ever increasing cracking variants (Section 10.1), and over joins to perform adaptive denormalization (Section 10.2).

In this direction, we believe that instrumentation-enabled engines can be instrumental in the space of adaptive physical database design by allowing us to piggyback physical database design choices within query execution in arbitrarily complex ways.

## 13.6 Online Query Optimization

Online query optimization is a domain where techniques recognize the fact that a database optimizer may decide on suboptimal plans due to the absence of detailed statistics or unawareness of runtime events. To address this problem, online query optimization techniques collect knowledge about a query during its execution (e.g., statistics of CPU and memory consumption or better selectivity and cardinality estimates) and observe runtime events (e.g., an increase in a memory budget). Based on this knowledge, they then make decisions on how to change a physical plan. As query complexity increases, such techniques require to collect and induce more sophisticated knowledge to reoptimize a query online. In Chapters 1 and 12, we discussed how recent complicated query optimization techniques including Probabilistic Predicates [LCKC18; LKC18], Smooth Scan [BGIA$^+$18], Sideways [IT08] and Lookahead [PDZ$^+$18] Information Passing, and Adaptive Joins [SQL18; Ora17] can use the underlying mechanisms of our physical plan instrumentation mechanisms to implement their logic. As such, and given recent advances in online query optimization (primarily using reinforcement and deep learning [KYG$^+$18; MP18]), we believe that instrumentation-enabled engines can provide the principle underlying mechanisms to ease the implementation and overall optimization of online query optimization techniques.

## 13.7 Interactive Data Profiling

We conclude this chapter by describing related work on (interactive) data profiling. As we noted in Chapter 9, data profiling is a domain that studies the statistics and quality of datasets (e.g., constraint checking; data type extraction; or key identification) while interactive data profiling allows users to interactively profile and examine the reasons for these results. For a classification of (interactive) profiling tasks and their connections with other domains including query optimization refer to [Nau14]. Recent profiling systems include extensible data profiling platforms (e.g., METANOME [PBF+15]), data wrangling and cleaning tools (e.g., Wrangler [KPHH11], Profiler [KPP+12], and NADEEF [EEI+13]), and user-guided functional dependency (FD) miners (e.g., UGUIDE [TBEO+17]). In Chapter 9, we showed how to evaluate data profiling tasks (i.e., functional dependency, uniqueness, and mismatch checks) and explore their results in instrumentation and provenance terms. Experimentally, we showed that SMOKE is capable of optimizing the evaluation and exploration of these tasks in comparison to their hand-tuned implementations within alternative profiling systems.

In this direction, we believe our results show evidence that instrumentation- and provenance-enabled engines provide a principle way towards the optimization of the important domain of (interactive) data profiling. Furthermore, our techniques show best practices for the interactive data profiling domain that profiling platforms, such as the ones described above, can exploit to further their expressive and optimization power.

# Chapter 14

# Conclusions

In Chapter 1, we posed two main classes of research questions that we aimed to address throughout this dissertation. We can summarize these questions as follows: (1) what are the mechanisms to facilitate the development of applications that operate over how queries are executed by databases and (2) provided a database engine augmented with such mechanisms, what is its overall expressive and optimization power.

To address the former class of questions, we introduced a database engine, namely, SMOKE, that exposes mechanisms in the form of a Physical Plan Instrumentation Framework. Our Physical Plan Instrumentation Framework comprises several components including Points that are used in the development of Instrumentors for pushing external logic within the logic of physical operators; a Scheduler that allows applications to schedule their instrumentation logic relative to the query execution; a Storage Manager that allows applications to access the state of the database, implement their own logic, and access the internal state of operators in a plan; an Announcer that allows applications to specify run time conditions and get notified when such events are met; and an Actions component that allows applications to modify, add, remove, and replace physical operators. In support of such mechanisms, we also outlined and addressed several technical challenges behind each component. Furthermore, we outlined the changes that SMOKE undertook in support of such mechanisms involving primarily its Compiler and the changes in its physical algebra.

Overall, we view our mechanisms and the underlying changes that we made to the database engine as the first step towards instrumentation-enabled database engines that can best facilitate the development of applications, across numerous domains, that rely their logic on how queries are executed by databases.

To address the second class of questions, we introduced and experimented with several instrumentation-based techniques on top of SMOKE across domains (i.e., positive and negative provenance management, interactive visualizations, interactive data profiling, physical database design, query discovery, online query optimization, and interactive applications). Expressiveness-wise, we showed how to express well-known techniques, introduce novel semantics on well-known techniques, and introduced novel techniques across domains. Performance-wise, we introduced techniques that are either on par with or several orders of magnitude faster than state-of-the-art hand-written alternatives. Finally, throughout our discussion, we introduced design principles and best practices for instrumentation in SMOKE. We believe our techniques and experimental evidence highlight the expressive and optimization power that instrumentation-enabled engines can provide across domains.

# Chapter 15

# Future Work

Throughout this dissertation, we took the first step towards the introduction of physical plan instrumentation mechanisms and associated instrumentation-enabled techniques from within our prototype database. We believe there is ample space for future work towards research and practice. Next, we summarize interesting future directions.

## Instrumentation

We start our discussion with future directions on instrumentation (i.e., extending support on alternative database designs, declarative specification of the instrumentation logic, and interoperation of instrumentation applications with user programs).

### Alternative Database Designs

SMOKE is an in-memory query compiled engine with limited or no support for columnar execution, vectorization, parallelization, distributed execution, interpretation, and on-disk storage. Introducing such, loosely speaking, features in a database that natively supports instrumentation needs to be followed by a revisit of the instrumentation-based mechanisms that we proposed in this dissertation. This is because such features result in different implementations of physical operators and internal components. Different implementations of physical operators and internal components, however, need to be followed by a revisit of our instrumentation mechanisms and their implementation (e.g., new physical operators require the introduction of instrumentation points and their semantics on processing data flows while the introduction of interpretation needs to be followed by changes on the design of instrumentors and internal database components, as we discussed in Section 6.10). Given the vast space of physical algebras, storage layouts, and query execution mechanisms that have been introduced over the years, extending support of our instrumentation mechanisms to alternative designs remains an interesting future work.

**Declarative Specification of Instrumentation Logic**

Another important direction for instrumentation-enabled engines, also connected with our discussion above, regards the declarative specification of instrumentation logic.

First, having to deal with the semantics of every possible underlying physical algebra may soon become bewildering as a research task. In this direction, note that the instrumentation points that we introduced in Section 6.3 have a logical underpinning. For instance, no matter whether a selection is performed in a columnar or row-major fashion there is always going to be a point in the logic of the selection where its parent consumes its results. Similarly, there are always going to be points in the selection that we can introduce negative data flows and there are always going to be points where we evaluate predicates. Hence, the points $\sigma_P^{\texttt{before}}$, $\sigma_P^{\texttt{after}}$, $\sigma_N$, and the techniques of the Actions component for modifying CNFs have logical underpinnings that are not strongly tied to the physical implementation. Building on these logical underpinnings, we believe there is a chance for a logical instrumentation algebra that can be compiled down to Instrumentors. In this way, we can allow different databases with different physical algebras to communicate their instrumentation logic, similarly to how a relational query can be equally expressed in different databases.

Second, recall from our discussion in Section 6.4.3 that SMOKE allows the declarative specification of instrumentation logic over points based on SQL. In this direction, we believe there is ample space for future work that allows other forms of declarative specification.

More specifically, following recent and older proposals for the declarative specification of physical database designs [IZH$^+$18; TSI96], we believe that the declarative specification of instrumentors to perform physical database design is an interesting future work, especially given the connections of instrumentation with adaptive physical database design that we showed in Part II. Similar we believe is the case for online query optimization techniques and their declarative specification through instrumentation applications. In fact, the `on` statement of the Announcer, that is strongly connected to how online query optimization techniques can be triggered for re-optimization purposes, already has a declarative form which can be read as follows: when a condition is met apply a resolution function. Specifying the actual runtime conditions and their resolution functions in a declarative form is an open challenge given the multiple different forms that runtime statistics and resolution functions can take across hardware specifications.

### Interoperation of Instrumentation Applications with User Programs

Our focus in this dissertation has primarily been on the specification of instrumentation applications right before query execution. User programs, however, may want to dynamically control instrumentation applications during and after the query execution. In such scenarios, an important direction for future work is on the interoperation of instrumentors with the runtime of user programs. Similarly to how databases provide SQL and database connectivity mechanisms (e.g., JDBC or ODBC) as interfaces to external programs for querying purposes, instrumentation applications should also provide interfaces to user programs. For instance, a program that issues a natural join query $\mathrm{A} \bowtie \mathrm{B}$ to a database may want the results of the query as well as the time spent on joining each tuple from $\mathrm{B}$. Accessing the timing information requires user programs to have programmatic access to instrumentors which is a non-trivial task. For instance, user programs may want to access this timing information online (e.g., get the timing information as long as it is available without waiting for the whole query to complete its execution). Furthermore, user programs may want to modify instrumentors at runtime. For instance, a user program may decide at runtime (i.e., after it has observed some timing results) that timing is required only for tuples with certain attributes or timing information should be aggregated over batches of tuples. As such, exposing optimized interfaces for the seamless interoperation of user programs and instrumentation applications is important to ease the implementation of user programs and the exposure of instrumentation-enabled engines.

Finally, throughout Part I we also proposed several guidelines on implementing instrumentation frameworks and provenance techniques on alternative databases designs, instrumentation security, and user experiences around instrumentation engines. To avoid redundancy, we omit further discussion here.

## Applications

Throughout our discussion on applications, we discussed several important task-specific future directions. For conciseness, here we summarize the two main directions based on which we proposed task-specific directions: expressiveness and optimization.

### Expressiveness

The techniques we have presented throughout cover only a limited class of functionalities across domains. Exploring the limits of instrumentation-enabled engines (i.e., recognizing what is expressible in instrumentation terms and what is not) across domains is an interesting future work. For instance, in the domain of interactive data profiling, we considered only FD, uniqueness, and mismatch checks. The class of interactive data profiling is much richer than only these checks (see [Nau14] for a recent survey) and understanding the extent to which instrumentation is the right way to go about addressing the exploration of data profiling is important. Overall, we believe this dissertation takes the first step towards understanding what tasks are candidates for instrumentation across application domains, yet understanding in complete the extent per domain is both an important and hard problem.

### Optimization

Regarding optimization, recall that our instrumentation- and provenance-based techniques across domains are over a fixed physical algebra. Changing the way a database performs query execution, as we discussed above, provides novel challenges (e.g., provenance capture under parallel query execution) that we have not addressed in this dissertation. Furthermore, for several tasks across domains, especially the ones covered in Chapters 7, 10 and 12, we considered how they can be expressed in instrumentation-enabled database engines and how we can introduce novel semantics through frameworks. Putting them into practice, extending their semantics, and using the proposed frameworks for optimization purposes in novel ways (e.g., devising algorithms for database cracking and adaptive denormalization based on online statistics or devising novel algorithms for crossfiltering across applications based on how users are currently interacting with the visual pane) remain open challenges that we have not addressed in this dissertation.

# Bibliography

[ABH$^+$13] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[ABS$^+$06] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, 2003.

[AC99] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. *SIGMOD Rec.*, 28(2):181–192, 1999.

[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[ADM$^+$15] Tim Althoff, Xin Luna Dong, Kevin Murphy, Safa Alai, Van Dang, and Wei Zhang. Timemachine: Timeline generation for knowledge-base entities. In *KDD*, 2015.

[Adv] AdventureWorks. https://msftdbprodsamples.codeplex.com/releases.

[AHS12] Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In *UIST*, pages 207–218, 2012.

[AIA14] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, 2014.

[AKLT15] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Val Tannen. Algorithms for provisioning queries and analytics. *arXiv preprint arXiv:1512.06143*, 2015.

[AW16] Daniel Alabi and Eugene Wu. PFunk-H: Approximate query processing using perceptual models. In *HILDA*, 2016.

[BC87] Richard A. Becker and William S. Cleveland. Brushing scatterplots. In *Technometrics*, 1987.

[BC07] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: A multidimensional workload-aware histogram. In *SIGMOD*, pages 211–222, 2001.

[BCHS17] Leilani Battle, Remco Chang, Jeffrey Heer, and Michael Stonebraker. Position statement: The case for a visualization performance benchmark. In *DSIA*, 2017.

[BCK$^+$11] Nicolas Bruno, Surajit Chaudhuri, Arnd Christian König, Arnd Christian König, Vivek Narasayya, Ravi Ramamurthy, Manoj Syamala, and Nico Bruno. Autoadmin project at microsoft research: Lessons learned. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, December 2011.

[BCS16] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *SIGMOD*, 2016.

[BCTV04] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, 2003.

[BGIA$^+$18] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth scan: Robust access path selection without cardinality estimation. *The VLDB Journal*, 2018.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. In *InfoVis*, 2011.

[Bru04] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[Bru09] Jake Brutlag. Speed matters for google web search (2009): http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[BZN05] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[CCD$^+$03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, 2003.

[CCT09] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. In *Foundations and Trends in Databases*, 2009.

[CDN11] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. An overview of business intelligence technology. *CACM*, 2011.

[CGS03] Surajit Chaudhuri, Prasanna Ganesan, and Sunita Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, 2003.

[CHZ$^+$08] Michael J. Cafarella, Alon Y. Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. Uncovering the relational web. In *WebDB*, 2008.

[CIOP14] Anup Chalamalla, Ihab F Ilyas, Mourad Ouzzani, and Paolo Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, 2014.

[CJ09] Adriane Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[CLKG16] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *SPE*, 2016.

[CLMR16] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. In *VLDB*, 2016.

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[CNR04] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *SIGMOD*, 2004.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. In *CACM*, 1970.

[cro15] Crossfilter. `http://square.github.io/crossfilter/`, 2015.

[CTV05] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: A post-it system for relational databases based on provenance. In *SIGMOD*, 2005.

[Cui01] Yingwei Cui. *Lineage tracing in data warehouses*. PhD thesis, Stanford University, 2001.

[CWH$^+$17] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. Data provenance at internet scale: architecture, experiences, and the road ahead. In *CIDR*, 2017.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 2000.

[DBCK17] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and TIm Kraska. Revisiting reuse in main memory database systems. In *SIGMOD*, 2017.

[DFG17] Daniel Deutch, Nave Frost, and Amir Gilad. Provenance for natural language queries. In *VLDB*, 2017.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[DIMT13] Daniel Deutch, Zachary G Ives, Tova Milo, and Val Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

[DKR97] Mark Derthick, John Kolojejchick, and Steven F. Roth. An interactive visual query environment for exploring data. In *UIST*, 1997.

[DRS$^+$05] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, 2005.

[EEI$^+$13] Amr Ebaid, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, Nan Tang, and Si Yin. Nadeef: A generalized data cleaning system. *PVLDB*, 2013.

[EU 18] EU GDPR. `https://www.eugdpr.org/`, 2018.

[Exc18] Excel. https://www.office.com/start/default.aspx, 2018.

[FKL$^+$17] Franz Faerber, Alfons Kemper, Per-Ake Larson, Justin Levandoski, Thomas Neumann, and Andrew Pavlo. Main memory database systems. *Foundations and Trends®in Databases*, 8(1-2):1–130, 2017.

[fol17] Facebook folly. `http://bit.ly/fbfolly`, 2017.

[FS15] Pedro Flemming and David Schwalb. Hyrisesql: A sql interface for hyrise a technical documentation. Technical report, Hasso-Plattner-Institute, 2015.

[GA09] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, 2009.

[GCB$^+$97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. In *Data Mining and Knowledge Discovery*, 1997.

[GCZ$^+$17] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. In *VLDB*, 2017.

[GJS74] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete problems. In *STOC*, 1974.

[GKIT07] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN*, 1982.

[GKM06] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, 2006.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[Goo18] Googlesheets. https://docs.google.com/spreadsheets/, 2018.

[GT17] Todd J. Green and Val Tannen. The semiring franework for database provenance. In *PODS*, pages 93–99, 2017.

[HA14] Jonathan Harper and Maneesh Agrawala. Deconstructing and restyling d3 visualizations. In *UIST*, 2014.

[Han12] Pat Hanrahan. Analytic database technologies for a new kind of user: The data enthusiast. In *SIGMOD*, 2012.

[HAW08] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. Generalized selection via interactive query relaxation. In *CHI*, 2008.

[HCDN08] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1):736–747, August 2008.

[HDBL17] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 2017.

[HGP03] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.

[HIKY12] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. In *PVLDB*, 2012.

[HKN$^+$16] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing google's datasets. In *SIGMOD*, 2016.

[HKW$^+$15] Daniel Haas, Sanjay Krishnan, Jiannan Wang, Michael J Franklin, and Eugene Wu. Wisteria: Nurturing scalable data cleaning infrastructure. *PVLDB*, 8(12):2004–2007, 2015.

[HMT11] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. Revisiting the partial data cube materialization. In *ADBIS*, pages 70–83, 2011.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. Discover: keyword search in relational databases. In *VLDB*, 2002.

[HS12] Jeffrey Heer and Ben Shneiderman. Interactive dynamics for visual analysis. *CACM*, 2012.

[HSG$^+$17] Joseph M Hellerstein, Vikram Sreekanti, Joseph E Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, et al. Ground: A data context service. In *CIDR*, 2017.

[IHW04] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 395–406, New York, NY, USA, 2004. ACM.

[Ike12] Robert Ikeda. *Provenance In Data-Oriented Workflows*. PhD thesis, Stanford University, 2012.

[IKM07a] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Database cracking. In *CIDR*, 2007.

[IKM07b] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[IMD] IMDB. http://www.imdb.com/interfaces.

[IMKG11] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. In *VLDB*, 2011.

[Int] Intel trace analyzer: https://software.intel.com/en-us/intel-trace-analyzer/documentation.

[IPC15] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, 2015.

[IPW11] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.

[IST$^+$15] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. In *VLDB*, 2015.

[IT08] Zachary G Ives and Nicholas E Taylor. Sideways information passing for push-style query processing. In *ICDE*, 2008.

[IW10] Robert Ikeda and Jennifer Widom. Panda: A system for provenance and data. *Data Engineering Bulletin*, 33(3):42–49, 2010.

[IZH+18] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *SIGMOD*, 2018.

[JCLJ08] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *MoBS*, 2008.

[JJHM14] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. M4: A visualization-oriented time series data aggregation. In *VLDB*, 2014.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming SQL standard. In *VLDB*, 2008.

[jQu] jQuery. `https://jquery.com/`.

[JRSS08] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[KAI17] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. Access path selection in main-memory optimized data systems: Should i scan or should i probe? In *SIGMOD*, pages 715–730, 2017.

[KBP+14] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. Rapid sampling for visualizations with ordering guarantees. In *arXiv*, 2014.

[KBY17] Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*, pages 1717–1722, 2017.

[KDCN11] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A statistical approach towards robust progress estimation. *Proc. VLDB Endow.*, 5(4):382–393, December 2011.

[KEA$^+$17] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, August 2017.

[KHDA12] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. Proton: multitouch gestures as regular expressions. In *CHI*, 2012.

[KIT10] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *SIGMOD*, 2010.

[KJTN14] Niranjan Kamat, Prasanth Jayachandran, Kathik Tunga, and Arnab Nandi. Distributed and interactive cube exploration. In *ICDE*, 2014.

[KLK$^+$18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 2018.

[KLZ13] Sven Köhler, Bertram Ludäscher, and Daniel Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*. 2013.

[KM05] Martin L Kersten and Stefan Manegold. Cracking the database store. In *CIDR*, 2005.

[KPHH11] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011.

[KPP$^+$12] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *AVI*, 2012.

[KYG$^+$18] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[LCKC18] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, 2018.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[LDY13] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable lineage capture for debugging disc analytics. In *SoCC*, pages 17:1–17:15, 2013.

[LGM$^+$15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 2015.

[LH14] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. In *Vis*, 2014.

[LJH13] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, 2013.

[LKC18] Yao Lu, Srikanth Kandula, and Surajit Chaudhuri. Interactive demonstration of probabilistic predicates. In *SIGMOD*, 2018.

[LKS13] Lauro Lins, James T Klosowski, and Carlos Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *EuroVis*, 19(12):2456–2465, 2013.

[LLG18] Seokki Lee, Bertram Ludäscher, and Boris Glavic. Pug: a framework and practical implementation for why and why-not provenance. *The VLDB Journal*, 2018.

[LLV] LLVM. `https://lldb.llvm.org/`.

[LLWZ07] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.

[LP14] Yinan Li and Jignesh M. Patel. Widetable: An accelerator for analytical data processing. *Proc. VLDB Endow.*, 2014.

[LR99] Zhe Li and Kenneth A. Ross. Fast joins using join indices. *The VLDB Journal*, 1999.

[LRB$^+$97] M Livny, R Ramakrishnan, K Beyer, G Chen, D Donjerkovic, S Lawande, J Myllymaki, and K Wenger. Devise: Integrated querying and visual exploration of large datasets (demo abstract). In *SIGMOD*, 1997.

[LSJM17] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. Adaptdb: adaptive partitioning for distributed joins. In *VLDB*, 2017.

[LWYZ16] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, 2016.

[Mal18] MAL Profilers. `https://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler`, 2018. Last accessed on 11/17/2018.

[May06] Marissa Mayer. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

[MC17] Rano Mal and Yul Chu. A flexible multi-core functional cache simulator (fm-sim). In *SummerSim*, 2017.

[MCACM17] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudre-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *CIDR*, 2017.

[MCF$^+$11] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 2011.

[MGMS10] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 2010.

[MGS11] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. Reverse data management. In *VLDB*, 2011.

[MGY15] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[MLVP14] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: Give me an example of what you need. *Proceedings of the VLDB Endowment*, 7(5):365–376, 2014.

[MMP17] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 2017.

[Mon15a] MonetDB. `https://www.monetdb.org/`, 2015. Last accessed on 11/17/2018.

[Mon15b] MonetDB Tachograph. `https://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler/tachograph`, 2015. Last accessed on 11/17/2018.

[MP18] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. *CoRR*, 2018.

[MSLN00] Ethan Millar, Dan Shen, Junli Liu, and Charles K. Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *J. Digit. Inf.*, 1(5), 2000.

[Mys18a] MySQL Performance Schema. https://dev.mysql.com/doc/dev/mysql-server/8.0.0/PAGE_PFS.html, 2018.

[Mys18b] MySQL Rewrite Plugin. `https://dev.mysql.com/doc/refman/8.0/en/rewriter-query-rewrite-plugin.html`, 2018. Last accessed on 11/17/2018.

[Nau14] Felix Naumann. Data profiling revisited. In *SIGMOD Record*, 2014.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[NKG$^+$17] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. Provenance-aware query optimization. In *ICDE*, 2017.

[NL14] Thomas Neumann and Viktor Leis. Compiling database queries into machine code. *Data Engineering Bulletin*, 37(1):3–11, 2014.

[NS00] Chris North and Ben Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *AVI*, 2000.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV)*, 2003.

[Ont] Airline On-Time Performance. `http://stat-computing.org/dataexpo/2009/the-data.html`.

[OOCR09] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[Ope] OpenFlights: Airport, airline and route data. `https://openflights.org/data.html`.

[Ora03] Oracle. Oracle database 10g: The self-managing database. Technical report, Oracle, 2003.

[Ora14] Oracle. Oracle endeca information discovery: A technical overview. Technical report, Oracle, 2014.

[Ora17] Oracle. Optimizer with oracle database 12c release 2. Technical report, Oracle, 2017.

[PBF$^+$15] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. Data profiling with metanome. In *VLDB*, 2015.

[PDCC15] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. S4: Top-k spreadsheet-style search for query discovery. In *SIGMOD*, 2015.

[PDZ+18] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow.*, 2018.

[Phy] Physician Compare National. `https://data.medicare.gov/data/physician-compare`.

[PIM15] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, 2015.

[Pos13] Hooks in PostgreSQL, 2013.

[Pow18] Power bi. `https://powerbi.microsoft.com`, 2018.

[PPI+14] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin Kersten. Database cracking: Fancy scan, not poor man's sort! In *DaMoN*, 2014.

[Pro] SQL Server Profiler. http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/sqlops5.mspx.

[PSSC17] Cıcero AL Pahins, Sean A Stephens, Carlos Scheidegger, and Joao LD Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. In *TVCG*, 2017.

[PSWC17] Marianne Procopio, Carlos Scheidegger, Eugene Wu, and Remco Chang. Load-n-go: Fast approximate join visualizations that improve over time. In *DSIA*, 2017.

[PW18] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained lineage at interactive speed. *ArXiv e-prints*, abs/1801.07237, 2018.

[QCJ12] Li Qian, Michael J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

[qt] Cross-platform software development for embedded and desktop. `https://www.qt.io/`.

[RAK$^+$17] Sajjadur Rahman, Maryam Aliakbarpour, Ha Kyung Kong, Eric Blais, Karrie Karahalios, Aditya Parameswaran, and Ronitt Rubinfield. I've seen enough: incrementally improving visualizations to support rapid decision making. *Proceedings of the VLDB Endowment*, 10(11):1262–1273, 2017.

[RCIR17] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. In *VLDB*, 2017.

[rea] React. `https://jquery.com/`.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *GPCE*, 2010.

[ROS15] Sudeepa Roy, Laurel Orr, and Dan Suciu. Explaining query answers with explanation-ready databases. In *VLDB*, 2015.

[RSt16] Rstudio shiny. https://shiny.rstudio.com/, 2016.

[RVL$^+$97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *USENIX Windows NT Workshop*, 1997.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[SCC$^+$14] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In *SIGMOD*, 2014.

[SDL18] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. Adaptive adaptive indexing. In *ICDE*, 2018.

[SE94] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *PLDI*, pages 196–205, 1994.

[SGB$^+$18] H. Strobelt, S. Gehrmann, M. Behrisch, A. Perer, H. Pfister, and A. M. Rush. Seq2Seq-Vis: A Visual Debugging Tool for Sequence-to-Sequence Models. *ArXiv e-prints*, 2018.

[Shn84] Ben Shneiderman. Response time and display rate in human performance with computers. In *CSUR*, 1984.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Symposium on Visual Languages*, pages 336–343, 1996.

[Sin01] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[SJD13] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *Proc. VLDB Endow.*, 7(2):97–108, October 2013.

[SJLM16] Anil Shanbhag, Alekh Jindal, Yi Lu, and Samuel Madden. Amoeba: a shape changing storage system for big data. In *VLDB*, 2016.

[SJTDP11] T. Smith, W. Johnson, R. Tamm-Daniels, and S. Probstein. Querying joined data within a search engine index. *US Patent No. 8073840*, 2011.

[SK98] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *VLDB*, pages 19–28, 2001.

[SMWH17] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. In *TVCG*, 2017.

[Spa18] Spark Monitoring and Instrumentation. https://spark.apache.org/docs/latest/monitoring.html, 2018. Last accessed on 11/17/2018.

[SQL18] SQLServer2017. Adaptive query processing in SQL databases. `https://docs.microsoft.com/en-us/sql/relational-databases/performance/adaptive-query-processing` , 2018.

[SRHH15] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. In *InfoVis*, 2015.

[SSD18] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643*, 2018.

[Sta86] Richard Stallman. `https://www.gnu.org/software/gdb/`, 1986.

[SVK$^+$07] Carlos E Scheidegger, Huy T Vo, David Koop, Juliana Freire, and Cláudio T Silva. Querying and creating visualizations by analogy. In *TVCG*. IEEE, 2007.

[tab] Tableau. `http://www.tableausoftware.com`.

[TBEO$^+$17] Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, and Nan Tang. Uguide: User-guided discovery of fd-detectable errors. In *SIGMOD*, pages 1385–1397, 2017.

[Ten16] Tensorboard: Visualizing learning. `https://www.tensorflow.org/guide/summaries_and_tensorboard`, 2016.

[TER18] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *SIGMOD*, 2018.

[Tid] Tidyverse. `https://www.tidyverse.org/`.

[TSI96] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The gmap: A versatile tool for physical data independence. *The VLDB Journal*, 1996.

[TSW11] Tomasz Tylenda, Mauro Sozio, and Gerhard Weikum. Einstein: Physicist or vegetarian? summarizing semantic type graphs for knowledge discovery. In *WWW*, 2011.

[Tuk77] John W Tukey. *Exploratory data analysis*. Reading, Mass., 1977.

[TXS$^+$15] Pawel Terlecki, Fei Xu, Marianne Shaw, Valeri Kim, and Richard Wesley. On improving user response times in tableau. In *SIGMOD*, 2015.

[TZW$^+$17] Ronny Tschüter, Johannes Ziegenbalg, Bert Wesarg, Matthias Weber, Christian Herold, Sebastian Döbel, and Ronny Brendel. An llvm instrumentation plug-in for score-p. In *LLVM-HPC*, 2017.

[USC] TIGER Products - Geography - U.S. Census Bureau. https://www.census.gov/geo/maps-data/data/tiger.html.

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems (TODS)*, pages 218–246, 1987.

[VAPGZ17] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, 2017.

[WBM14] Eugene Wu, Leilani Battle, and Samuel R. Madden. The case for data visualization management systems: Vision paper. In *VLDB*, 2014.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

[WHW16] Yifan Wu, Joseph M Hellerstein, and Eugene Wu. A devil-ish approach to inconsistency in interactive visualizations. In *HILDA*, 2016.

[Wic09] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009.

[Wid05] Jennifer Widom. Trio: a system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[Wil03] Adalbert Wilhelm. User interaction at various levels of data displays. In *CSDA*, 2003.

[WLPS17] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, 2017.

[WM13] Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[WMS12] Eugene Wu, Samuel Madden, and Michael Stonebraker. A demonstration of dbwipes: clean as you query. In *VLDB*, 2012.

[WMS13] Eugene Wu, Samuel Madden, and Michael Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*, pages 865–876, 2013.

[WPM$^+$17] Eugene Wu, Fotis Psallidas, Zhengjie Miao, Haoci Zhang, Laura Rettig, Yifan Wu, and Thibault Sellam. Combining design and performance in a data visualization management system. In *CIDR*, 2017.

[WS97] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.

[XZAT18] Jane Xu, Waley Zhang, Abdussalam Alawini, and Val Tannen. Provenance analysis for missing answers and integrity repairs. *IEEE Data Eng. Bull.*, 41(1):39–50, 2018.

[YKSJ07] Ji Soo Yi, Youn ah Kang, John Stasko, and Julie Jacko. Toward a deeper understanding of the role of interaction in information visualization. In *TVCG*, 2007.

[ZPSP17] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *Proc. VLDB Endow.*, 2017.

[ZSF17] Zhao Zhang, Evan R Sparks, and Michael J Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In *HPDC*, pages 143–153, 2017.