ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 263-0800-00L

Systems Group, Department of Computer Science, ETH Zurich

Optimising Distributed Dataflows in Interactive Environments

by
Nikolas Göbel

Supervised by
Frank McSherry, Gustavo Alonso

November 2018 - April 2019

inf | Informatik
Computer Science

Contents

1 Abstract

The capability to maintain sophisticated relational queries over potentially unbounded data streams is a critical competitive advantage for data-driven organizations. Traditional approaches to relational query optimization assume ad-hoc execution, benevolent distribution of data, and the availability of accurate statistics at planning time. Established practice in low-latency stream processing assumes comparatively simpler computational models and relaxed consistency guarantees.

Differential computation generalizes incremental computation to support data-parallel relational operators and arbitrarily nested, least fixed point iteration. *3DF* utilizes differential computation to provide continuous, incremental evaluation of Datalog queries over data streams. In this work we identify the challenges to providing these new capabilities to dynamic multi-user environments and describe solutions adapted from the database and stream processing communities.

We implement our findings in 3DF and *Differential Dataflow*, an implementation of differential computation. We evaluate the resulting system and showcase in particular (I) the use of delta-queries to maintain joins of arbitrary arity with a constant memory footprint, (II) the use of worst-case optimal join algorithms to provide predictable query performance on adversarially skewed data, and (III) strategies to share resources between clients without giving up individual optimization opportunities.

Acknowledgements

I would like to thank Professor Gustavo Alonso for introducing me to Timely and Differential Dataflow, and for supporting my work on these systems.

I am beyond grateful to Dr. Frank McSherry for condensing a decade of learning into mere months, and for expanding my views on system design.

I would also like to thank Andrea Lattuada for his support, and in particular for his work on monitoring Timely computations.

2 Introduction and Motivation

As more and more industries adopt data-driven decision-making and companies are increasingly looking to provide digital services, the capability to extract actionable insights from many continuous data streams in a timely manner is becoming a critical competitive advantage.

From an end user perspective, any results derived from the data flowing through an organization should above all be correct, incorporate the most recent data available, and reflect a consistent view of the organization as of some point in time. Additionally, it is not acceptable any longer to hold information in mutable cells. Rather, information must be recorded in such a way that historical states can be recovered for analytical and auditing purposes.

To that end, many companies are starting to adopt event-driven architectures in which data producers and data consumers are uncoupled via a shared, append-only log of records. Services coordinate reactively, via the arrival of new data. While this approach makes for understandable, flexible systems, it puts a larger burden on data consumers to construct and *maintain consistent, up-to-date views* on the subset of information that is relevant to them.

Purpose-built stream processing systems can meet very high throughputs and near real-time latencies on this task, but lack support for expressive programming models and strong consistency guarantees. This puts a significant burden on application developers. Specialized graph-, OLAP-, and time-series databases on the other hand support strong consistency and complex queries, but their ad-hoc interaction models do not fit well into reactive, near real-time environments.

Only recently have dataflow systems emerged that support strongly consistent, incremental maintenance of complex computations over high-throughput data streams. The resulting systems respond efficiently to unbounded, arbitrarily changing inputs, support distributed execution out-of-the-box, and thus provide a foundation onto which data consumers can effectively off-load view maintenance. These systems therefore sit at the intersection of databases and stream processing, and thus merit re-visiting established practice in both fields.

3 Problem Description and Outline

Throughout this work we assume a setting in which clients arrive and leave dynamically. Clients register relational queries and expect result sets to be maintained in near real-time, as data from multiple heterogeneous sources streams into the system.

We are therefore tackling the problem of multi-query optimization, which is distinct from single-query optimization inasmuch as we do not concern ourselves with producing optimal execution strategies for any single query. Rather, we want to *limit the number of disastrous plans* picked, because those have an outsized negative effect on *all* users of the system. Our priorities are as follows:

1. **Robustness** Above all we would like for the resulting system to match user expecations, and to perform consistently well and in a predictable manner.

2. **Multi-user Support** In order to be useful as a general-purpose system, we must support many concurrent use cases and clients — be they analysts working with interactive sessions or distributed, long-running services processing large amounts of information.

3. **Low-latency and High-Throughput** The attractiveness of stream processing derives from its performance characteristics, and the possibility to obtain results in a timely manner. While aiming for (1) and (2), we must take care to overall maintain these properties as much as possible.

To this end, we will in section 4 first provide some background on the data-parallel dataflow model, modern stream processors, and differential computation. Section 5 describes 3DF, a streaming relational query engine we've built in previous work, that will serve as a baseline and reference implementation. 3DF enforces a highly opinionated data model that fundamentally affects all other considerations made throughout this work. We motivate these opinions and provide a short overview of the state of 3DF before this work.

Section 6 is a survey of existing optimization strategies from both the database and the stream processing communities, and a discussion of their applicability to our setting.

Across the subsequent chapters we then cover what we have found to be the most significant obstacles that a system such as 3DF must overcome in order to satisfy the robustness, usefulness in multi-user environments, and performance properties identified in the previous chapter. These are (in no particular order): the over-eager implementation of queries in a dynamic environment (section 7), the explosion of intermediate state produced by high-arity joins (section 8), the robust estimation of join cardinalities (section 9), and navigating the trade-off between highly generalized plans whose resources can be shared between many users and highly specialized plans that can exploit additional optimization opportunities (section 10).

For each of these we describe the problem in general terms, provide specific examples, propose strategies to remediate them, and evaluate our implementation of these proposals. Implementation details on all techniques discussed are provided later in section 12.

Section 11 provides a similar analysis of the adverse effects of non-uniformly distributed workloads and how certain aspects of relational operators and 3DFs data model might exacerbate this problem. While we do propose solutions, these are not implemented in 3DF at the time of this writing.

We close with a discussion of less significant challenges discovered over the course of this work, as well as of opportunities for further work in section 14, and summarize our contributions in section 15.

4 Background

3DF builds on a rather recent body of work that allows us to even consider the continuous maintenance of queries, whose complexity used to be the exclusive domain of analytical databases. In this chapter we describe the dataflow model of computation as a foundation for stream processing systems. We then provide intuition for the concept of *progress tracking*, and how it enables the efficient execution of nested, cyclic dataflows. Finally, we discuss how *differential dataflow* extends the dataflow model to *incremental* operators and enables recursive queries through independent, efficient iteration of regions of the dataflow.

4.1 Dataflow

Dataflow, as commonly used, refers to two different but related concepts, both of which are relevant to this work.

On the one hand, *Dataflow* is an architectural paradigm in which the execution of a computation is coordinated entirely via the availability of data. This is in opposition to the von Neumann architecture, which uses control structures (sequential statements, branching via conditionals, loops) to coordinate the execution. Today we often describe systems that follow this idea as *reactive*.

On the other hand, *Dataflow programming* refers to a programming paradigm in which the structure of a computation is expressed as a directed graph. This dataflow graph allows us to reason about the (potentially distributed) execution of the computation. In particular, it allows us to infer data dependencies and thus execution strategies that avoid recomputation of unaffected paths through the graph on new inputs. In the distributed setting, a fine-grained view of data dependencies allows us to exploit more opportunities for concurrent execution.

4.2 Progress Tracking

Timely Dataflow (McSherry (d)) is an implementation of the cyclical dataflow model introduced in Murray et al. (2013). It is written in the Rust language. Timely Dataflow follows a data-parallel approach, meaning each worker in a Timely cluster runs the exact same dataflow computation, with inputs partitioned between them.

Dataflow computations over unbounded, potentially out-of-order inputs must reason about times at which it is safe to produce results. Producing results over-eagerly will keep latencies low, but causes downstream inconsistencies, due to forwarded results not incorporating all relevant inputs. Likewise, holding results back for too long ensures correctness, but increases latencies unnecessarily. Taken to the extreme, this end of the spectrum corresponds to batch processing.

To solve this problem, a logical timestamp is attached to each datum and an order is imposed on them. The ordering controls visibility, thus detaching physical from logical availability. Taken together, propagating information

about input epochs along the dataflow graph allows us to infer the set of logical timestamps for which it is safe to produce results.

Progress tracking is the problem of determining for each point in the dataflow the set of input timestamps that may still be received there. This set is represented by a *frontier* consisting of one or more mutually incomparable times. Inputs might still be received for any time in advance of any of the frontier elements. Conversely, results for times earlier than the frontier can be propagated safely. Timely Dataflow is a runtime for data-parallel dataflow computations that coordinate via fine-grained progress tracking.

Without a coordination mechanism like progress tracking, query results will in general not reflect a consistent logical time. While highly undesirable from a business point of view, this can also cause non-terminating iterative computations.

4.3 Differential Dataflow

As the complexity of our computation grows and data volumes increase, we will want to avoid recomputing results from scratch whenever an input changes. Rather, we would like to incrementally update previous results with (hopefully) small changes. In particular, we would like to do so even for iterative computations, which arise in the evaluation of recursive queries.

This model of computation is called *incremental computation*. In order to incrementalize a computation $f(X) = Y$ on some collection X we must find a corresponding δf s.t. $f(X + \delta x) = f(X) + \delta f(\delta x) = Y + \delta y$. To be of practical use, we require δf to only perform work proportional to the size of δx. This can often be achieved whenever small input changes cause small output changes, as compared to the size of the entire collection. In these cases we can hold on to the latest version of the output collection Y and use it to more efficiently compute δy.

In a distributed setting with iterative computations, incremental computation faces a significant challenge because we must distinguish the *latest version* of a collection under two possible sources of input (one from upstream and one from iterative feedback). Assigning totally ordered timestamps prevents us from computing iterations for batches of inputs in parallel, effectively serializing the computation.

Differential computation (McSherry et al. (2013)) generalizes incremental computation by allowing timestamps to be partially ordered. This allows us to represent multiple different sources of inputs along different coordinates in a multi-dimensional timestamp.

Differential Dataflow (McSherry (b)) is a programming framework implementing differential computation on top of Timely. It allows for the expression of data-parallel dataflow programs as functional-relational transformations over *collections* of data, using incrementalized implementations of well-known operators such as `map`, `filter`, `reduce`, and `join`. Any region of a differential dataflow can be independently iterated to fixed point. The entire computation

responds efficiently to arbitrary changes in its inputs. Collections implement multiset semantics.

Finally, the dynamic environments we are targeting necessitate the creation of new dataflows at runtime. New query dataflows will often require access to inputs that predate their creation, in order to give correct initial result sets. Having to retrieve historical inputs from external input sources would incur not just significant latencies, but also cause much tighter coupling than simple change data capture.

Differential Dataflow has first-class support for compact in-memory representations of historical collection traces, via so called *arrangements*. Arrangements maintain compacted, indexed batches of updates to a collection. The resulting indexed state can be shared between operators, and allows for the efficient, dynamic creation of dataflows that can feed off of them.

To our knowledge, Differential Dataflow is the only system that can (a) maintain the performance characteristics of stream processors for complex relational computations, as required by recursive SQL queries or the evaluation of Datalog rules, and that (b) allows for low-latency, shared access to historical inputs.

5 Design and Implementation of 3DF

With a system like Differential Dataflow (*Differential*) available it becomes feasible to rethink the boundaries between the domain of analytical databases and that of near real-time stream processors.

In previous work (Goebel (2018)) we therefore built 3DF (short for *Declarative Differential Dataflow*), a dynamic query engine built on top of Differential. While Differential computations are compiled into static binaries, 3DF allows clients to register and unregister relational queries at runtime and to receive output changes via the network.

The existing 3DF codebase will act as baseline and reference implementation for later chapters. Thus, this chapter is dedicated to describe the design and implementation of 3DF at the outset of this work.

5.1 Data Model

3DF imposes strong opinions on data modeling. In particular, we adopt a unified, *attribute-oriented* data model. The fundamental unit of modeling are facts, represented by entity, attribute, value triples. Although conceptually we are modelling a single space of facts as it evolves through logical time, we keep track of individual attributes in their own Differential collections. Attribute collections are thus binary relations. Throughout this work we use the terms *attribute* and *base relation* interchangeably. We use names starting with a colon to identify attributes (e.g. `:person/name`, `:comment/parent-post`).

An attribute-oriented data model corresponds to a fully normalized relational model, and is heavily inspired by graph databases, RDF, and research on triple stores. 3DF also draws heavily from database systems like Datomic (Hickey (2013)) and LogicBlox (Aref et al. (2015)).

We believe that important semantics — such as schema, access and privacy policies, or compaction intervals — must be configurable at the level of an individual attribute, rather than in aggregate (e.g. on a relation in a traditional database schema), to satisfy the need for evolvable information systems in the real-world. Other organizational aspects such as documentation and namespacing are also most naturally expressed at this fine-grained level. An important effect of attribute-oriented modeling is the ability to source attributes from different external systems even though they might belong together from a domain modeling perspective.

It is well established that data processing systems with data models optimized for a specific domain can outperform general-purpose systems by orders of magnitude. Today, users can (in addition to traditional RDBMS) choose from column-stores designed for analytical queries, databases for time-series, graphs, or geographical data, or full-text search engines. We share the view of Aref et al. (2015) towards specialized databases. Their use is warranted in scenarios where they can provide at least an order of magnitude in efficiency gains. Below that, the cost of developing, maintaining, and integrating a specialized system

dominates. We suspect that this threshold is even higher for many industrial users with no in-house competence in designing data-processing systems.

Another important characteristic of modern analytical tasks is the prevalence of categorical features. In a relational setting, features correspond to distinct unary relations. In a traditional, row-oriented data model, modeling such data will lead to very wide, sparse tuples. Attribute-oriented data models can represent such data much more efficiently, and materialize only necessary attributes, at the cost of frequent multi-way joins across relations of potentially highly varying selectivities. These same insights drive the design of modern, column-oriented analytical databases. We use the terms *attribute* and *column* interchangeably throughout this work.

We therefore think it justified to fix attribute-orientation as a strong requirement for 3DF, and investigate instead whether novel approaches in other areas (such as Differential Dataflow itself) allow us to build systems that are both useful in real-world settings, and still offer competitive performance across a wide class of use cases.

5.2 Query Plan Language

3DF itself does not enforce a specific query language, but is intended to support relational query languages in general. We designed 3DF specifically with the needs of a reactive Datalog engine in mind. Datalog supports recursive rules and is thus well suited to expressing subpattern queries on graphs, large n-way joins, and many other common, relational computations in a concise way. The base language can be easily extended into various domains (e.g. temporal patterns, or probabilistic queries).

Datalog and other relational languages (such as SQL and its dialects) relies on a small set of computational primitives: selections, joins, projections, and set union. Datalog draws its expressive power from recursive rules, which can be supported by iterating the query evaluation to fixed point. With each new iteration, facts derived in the previous evaluation are incorporated. The computation stops when it reaches a fixed point, i.e. when no new facts are derived in an iteration. On top of those primitives, we support stratified logical negation (*negation as set difference*).

Differential comes with a built-in two-way `join` operator. Selections correspond to Differential's `filter` operator. Set union is expressed as `concat`. Projections are implemented as a `map` extracting the requested offsets from a stream of tuples. Set difference is implemented via the `antijoin` operator.

Timely and Differential support local iteration to fixed point, meaning parts of the dataflow graph can be iterated independently from one another and from the overall computation itself. Complex iteration patterns, such as mutual recursion, can be expressed by means of Differential's `Variable` abstraction. Variables represent collections that are not completely defined until the entire dataflow is constructed.

At the outset of this work, 3DF enforced set semantics over Differential's multisets, via use of the `distinct` operator. This changed over the course of

this work and multiset semantics are now reflected as is. The reasons for this are detailed later in subsection 14.5.

Within 3DF, query plans are represented by trees of operator descriptions (*plan stages*). In addition to plan stages for the operators described above, we provide stages for various aggregations on tuples, as well as data transforms and basic arithmetic via built-in functions.

The leaf nodes of a query plan are made up out of *data patterns*, which encode the supported access paths for facts:

- **MatchA := [?e a ?v]** reads an entire attribute collection a.
- **MatchEA := [e a ?v]** reads all values associated with entity e for attribute a.
- **MatchAV := [?e a v]** reads all entity ids which have the value v associated with them for attribute a.

This concludes our short summary of 3DF and its foundational technologies. We've covered how 3DF provides a baseline of dynamic, relational capabilities within an incremental, streaming setting. We also gave arguments for 3DF's somewhat extreme stance on data model flexibility and evolvability.

6 Known Techniques and Related Work

Over the previous chapters we have given arguments for why Differential Dataflow poses a unique opportunity to build a system that combines the performance characteristics of distributed stream processing with the analytical power of relational and graph databases. We have also introduced 3DF as a prototype of such a system.

In this chapter we will survey the decades of research on optimizing both relational query processing and streaming computations, and discuss to what extent these techniques can aid us in making 3DF fit for use in highly dynamic, multi-user environments.

6.1 Cost-Based Query Planning

The field of relational query planning is vast. Query planning is the problem of, given a tree structure of relational operators, determining the most efficient order in which to execute these operators without affecting the result set.

Selinger et al. (1979) give the canonical query planner, split into three components: cardinality estimation, plan space enumeration, and a cost model. This seminal work, done for the System R DBMS, first established the practicality of declarative query languages and — to this day — serves as the reference architecture for many modern databases in productive use.

Selinger et al. introduce the concept of an *access path*. Access paths represent the different ways of accessing the tuples in a relation. The simplest access path is a full scan of the relation, but various indices might be available as well.

For queries that don't involve joins, the System R planner employs a cost model incorporating statistics on the expected selectivity of each access path available to service each of the requested selection predicates. Additionally, the planner will check whether the desired output ordering matches the order provided by any suitable access path.

The complete cost model is somewhat more complex, because it allows for a weighting between I/O and CPU cost, and because it must deal with the fact that memory segments will contain tuples from different relations. Therefore the cost of scanning through a seemingly small relation might be much higher in reality. Given this cost model, the planner can pick the cheapest access path to service the query.

Of course the more interesting case is that involving multiple relations, which must be joined together. Given the cost model for queries on individual relations, the planner now has to enumerate join order permutations, compute the overall cost, and pick the cheapest overall plan. Simple heuristics are used to reduce the number of permutations under consideration.

Many variants and improvements on this architecture have been proposed over the years, most notably so in a long line of research by Graefe et al. (Graefe and DeWitt (1987), Graefe and McKenna (1993), and Graefe (1995)). These architectures vary in their plan enumeration strategies (bottom-up vs top-down), and improve the extensibility of the planning architecture. They share however

the basic approach of heuristics-based cost estimation in combination with plan enumeration.

6.2 Cardinality Estimation

> The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. Everything in cost estimation depends upon how many rows will be processed, so the entire cost model is predicated upon the cardinality model.
>
> Lohman (2014)

It is well established, that cardinality estimation is both the most important part of cost-based query planning, as well as the hardest to get right. Leis et al. (2015) show that estimation errors grow exponentially with the number of joins, for a representative selection of real-world, relational DBMS. They also show that correct cardinality estimates are much more important for plan performance than more accurate cost models. Due to 3DFs highly normalized data model, a large class of relevant queries require multi-way joins across many base relations. This implies that 3DF would be exceedingly sensitive to these kinds of ballooning errors.

Neither the importance of cardinality estimation, nor the difficulty of doing it accurately are surprising. Until recently, most strategies for planning multi-way joins consisted of breaking down a n-way join into a (hopefully optimal) sequence of two-way joins. In such a setting, accurately estimating the number of tuples contained in the result set of joining two relations is crucial, in order to pick a plan that materializes as few tuples as possible. Even slightly over-optimistic selectivity estimates can cause cardinality estimation errors proportional in the size of the input relations.

Many modern database systems still compute their estimates under the same assumptions originally proposed for System R.

- **Uniformity**

  Uniformity is the assumption that the values in an attribute are uniformly distributed. Under this assumption, the selectivity of predicates on any specific value is computed simply as the inverse of the domain cardinality of the attribute. This is of course a naive assumption, as attributes in the real world tend to exhibit a certain bias, e.g. age and location of users of a specific service.

- **Principle of Inclusion**

  Selinger et al. (1979) give the selectivity of a relational equi-join on two columns as the inverse of the greater of their domain cardinalities. This assumes that the domains of the two columns overlap. On otherwise unfiltered columns with appropriate referential constraints between them, this

assumption is valid. As soon as we apply other selection predicates before joining, this assumption will not hold in general.

- **Independence**

  Multiple conjunctive predicates on the same relation are assumed to be uncorrelated, allowing for their combined selectivity to be computed as the product of their individual selectivities. While at first glance, with the good relational practice of normalizing data models in mind, this seems like a reasonable assumption, it quickly breaks down on real-world data. A simple example might be the two attributes `:age` and `:salary`, which tend to correlate in most organizations. Breaking such functional dependencies down into more fundamental, independent attributes, and combining them with intricate models necessary to re-derive salaries, is not feasible in practice. Often relations are the most succinct and pragmatic way to model such functional dependencies.

- **Global Information**

  It is assumed that the query planner has access to up-to-date, accurate statistics on all relevant query inputs. This critical assumption does not hold in the continuous query processing setting, where not a single relevant input might have yet been seen at planning time.

While more sophisticated statistical models and methods have been proposed to loosen the reliance on assumptions of uniformity and independence (Markl et al. (2007), Ilyas et al. (2004)), it is the assumption of global information that prevents the use of traditional cardinality estimation when maintaining queries over potentially unbounded streams.

6.3 Adaptive Query Processing

In the previous section we have seen that traditional approaches to cardinality estimation (and thus to join ordering and query planning at large) are susceptible to exponentially compounding estimation errors, due to the use of simplistic heuristics, and furthermore can not be translated into the continuous query processing setting, due to the unavailability of *any* meaningful statistics.

An additional problem arises in this context, because even when accurate heuristics are available at query planning time, they might bear very little resemblance to those gathered after a few hours of new inputs. Therefore any planning decisions made based on those assumptions will have to be revised over time. In a general purpose, multi-user environment, queries might live anywhere from a few seconds to months or even years — ample time for the underlying distributions to change dramatically.

Adaptive query processing explores the idea that in environments where disastrous plans are to be expected every once in a while, and planning assumptions change constantly over time, systems should be able to adapt their planning decisions on the fly.

Avnur and Hellerstein (2000) introduce a new query processing architecture, called *eddy*, which "continuously reorders operators in a query plan as it runs". In a dataflow setting, this is best imagined as an additional dataflow operator routing input tuples between a set of candidate operators. The routing logic is then adapted, based on the measured flow of tuples from and to each of the candidate operators.

A significant challenge in Avnur and Hellerstein (2000) is the problem of identifying *moments of symmetry*, synchronization points within the processing pipeline, at which operator inputs can be reordered without the need to modify the accumulated operator state, which can be on the order of the entire input relation for joins. They analyse various families of join algorithms under this new criterium. Differential's incremental join operator is entirely coordinated by its inputs and, in combination with logical timestamps, might not necessarily be affected by the same considerations. However an exact translation of eddies into Differential Dataflow would require more investigation.

A complementary approach to the adaptive query processing problem is that of *progressive optimization*, as employed by Markl et al. (2004). This work extends the query processing pipeline with a reoptimization step, during wich cardinality estimates are validated against the actual values measured during execution of a previous query. Reoptimization imposes a synchronization barrier on the processing pipeline, making it inappropriate for long-lived, low-latency dataflows.

However, while the specifics of an adaptive query processing primitive would therefore likely be different in Differential Dataflow, the motivating observations clearly apply to our setting.

6.4 Worst-Case Optimal Join Processing

Taken to their logical conclusion, the ideas from adaptive query processing lead to the nascent field of worst-case optimal join processing. Here the fundamental observation is that the traditional approach to multi-way joins (consecutive two-way joins, or *join-at-a-time*) will always be sub-optimal for certain queries, because they generate intermediate results that can be larger than the final result set could ever be.

The initial work on worst-case optimal join algorithms is due to Ngo et al. (2012), with independent results by Veldhuizen (2012). Both of these algorithms (called *NPRR* and *Leapfrog TrieJoin* respectively) avoid generating intermediate results and are thus considered to process n-way joins all at once, instead of join-at-a-time. In fact, both were later generalized under a common framework in Ngo et al. (2013), called *GenericJoin*.

Given some ordered sequence of target variables and a set of relations, GenericJoin proceeds *variable-by-variable* rather than relation-by-relation, starting from the set containing only the empty prefix. From a set of prefixes, GenericJoin iteratively computes the *extension* of that set to the next variable in the variable ordering.

For each such extension stage, candidate extensions will be drawn from whichever relation promises to propose the least number of extensions, and intersected with the would-be proposals from all other relations. Assuming suitable indices on the relations, determining the number of proposals for a relation and a given prefix can be done in constant or logarithmic time. Similarly, intersecting a proposed extension with a relation's proposal index can be done in constant or logarithmic time. Additionally, proposals must be provided with constant delay.

Under these assumptions, each extension stage of a GenericJoin will only do work on the order of the largest possible result set. Picking suitable variable orderings is a crucial part of these new algorithms. While sub-optimal orderings do not affect the theoretical guarantees, they can make a significant difference in practice. In other words, while the number of materialized tuples might not exceed the worst-case bound on the size of the result set, that bound can still be to loose for good performance in practice.

Ciucanu and Olteanu (2015) apply ideas from the theory behind worst-case optimal join algorithms within the framework of *factorized databases*. They show that worst-case optimal complexity can be achieved with join-at-a-time plans as well, by eliminating redundancy from the representation of intermediate results.

More recently, Ammar et al. (2018) give worst-case optimal join algorithms for the data-parallel dataflow setting. The resulting operators are very much reminiscent of eddies, in that they make planning decisions on a per-tuple basis. They also extend the worst-case optimality guarantee to cover communication costs between workers.

Worst-case optimal join algorithms are therefore promising for two reasons. For one, they extend the class of queries for which we can hope to find non-disastrous plans, and for which no optimal two-way join ordering exists. Second, they implement a very granular, non-heuristical form of adaptive query processing without the need for synchronized re-optimization points. We make use of an adapted version of the worst-case optimal join algorithm by Ammar et al. (2018) in section 9, in order to provide predictable query performance without relying on heuristic cardinality estimation.

6.5 Continuous Query Processing

Continuously evaluating queries over data streams using incremental computation is of course not a new idea, and commonly known as *incremental view maintenance*. Early treatments such as Terry et al. (1992) identify the shortcomings of non-incremental, periodic execution of queries, propose continuous incremental maintenance as a solution, and identify interesting semantics aspects, such as the need for stratified negation. Their *Tapestry* system however supports only monotonic queries over append-only workloads on a single thread. Differential Dataflow provides the same incremental semantics, while supporting retractions and distributed execution.

Blakeley et al. (1986) introduce the concept of *differential re-evaluation* of views. In particular, they exploit the associativity of the join operator and

observe that for changes to a single relation, $A' \bowtie B = (A + \delta a) \bowtie B = (A \bowtie B) + (\delta a \bowtie B)$ — implying that the relational join operator is already incrementalized w.r.t. changes to any individual of its inputs.

Further, because of the distributivity of the join operator, the union of multiple such views can be constructed to arrive at an incrementalized join operation w.r.t. to arbitrary changes to *any* of its inputs.

We adopt this approach in section 8, in order to incrementally process high-arity joins with only a constant memory footprint in addition to the indexed base relations.

6.6 Stream Optimization

We turn now away from optimization as practiced by the database community, and survey the state of general stream processing optimizations. Hirzel et al. (2014) provide a catalog of the most common optimizations in stream processing, of which we will highlight a few.

- **Operator Reordering** is a much simpler version of the ideas discussed in the previous chapters. The basic idea is to move more selective operators upstream, in order to minimize the number of tuples propagated downstream. Selectivity in this setting is treated mostly at the level of operator types (e.g. `filter` has a higher selectivity than `map`).
- **Redundancy Elimination** attempts to remove redundant operators, e.g. by moving equivalent operators on two branches of a dataflow into their shared trunk.
- **State Sharing** avoids unnecessary copies of data, by allowing operators to access a shared piece of state, e.g. an index.
- **Batching** exploits the fact that many operators can amortize processing across many inputs.

An interesting trade-off between pushing selective operators upstream and redundancy elimination arises in the multi-tenant setting, as the more selective operators tend to differ between computations. Pushing these operators upstream might thus severely reduce the re-usability of a dataflow trunk, while significantly reducing the number of tuples flowing downstream. Hirzel et al. cite more sophisticated techniques to share dataflow elements in a multi-tenant setting. We will come back to these in section 10 to find strategies, that are not affected by this trade-off.

A similar trade-off arises between separation of operators into smaller computational steps in order to exploit pipeline parallelism and the merging of operators into larger units, in order to minimize communication and scheduling overheads. For example, multiple transform or filter stages can be merged into a single one by composing their transformation functions / predicates, and vice versa.

Most of the techniques explored in this line of inquiry are geared towards simpler computational frameworks than the relational model and its extensions. Additionally, Differential's arrangements already provide a general purpose way to share compacted and sorted batches of tuples between operators.

Therefore, while micro optimizations on the dataflow graph can be beneficial, we are more concerned with optimizations that affect the overall system design and have the potential to provide asymptotic improvements.

6.7 Rule Engines

Finally, we look into the intersectional topic of rule engines. Rule-based systems allow for the expression of business rules, often in a high-level language. Registered rules will then be continuously evaluated against an input stream of business events. It quickly becomes infeasible to re-evaluate all rules on every input. Specialized algorithms are employed to efficiently determine the subset of rules that might possibly be affected by a new input. Those rules are then re-evaluated.

The RETE algorithm due to Forgy (Forgy (1989)) is the archetypical pattern-matching algorithm powering rule engines. At its core, RETE is an algorithm to compile rule definitions into a corresponding *discrimination network*. This approach is very much reminiscent of a dataflow computation.

Most of the operators (called *nodes*) that Forgy describes are simple filters for an input's data type and similar discriminating attributes. These are easily replicated by Differential's `filter` operator. A clause such as `[?x :class/attr ?a]` will furthermore act as an inherent object-type filter. Forgy also describes `join` nodes, which are easily emulated by Differential's own `join` operator.

Critically, RETE networks assume that leaf nodes are traditional, ad-hoc queries run against a conventional database. In Differential, the entire computation is incrementalized. While the literature on rule engines does provide interesting approaches to incrementalizing traditional query engines, they lack the relational abstractions, strong consistency guarantees, and the ability to distribute recursive queries.

6.8 Conclusions

We reiterate our desire for a practical system serving diverse data needs in real-world organizations. Consequently, we care less about picking optimal queries for every individual query and are more interested in minimizing the number of disastrous plans chosen. Such plans not only affect individual query performance, but negatively impact all users of the system.

At a first glance, translating the System R approach to 3DF seems straightforward. Materialized sub-computations (via Differential's arrangements) could provide additional access-paths to tuples, for which a reasonable cost model could be established. Systems such as Condie et al. (2008) provide interesting precedence of applying Datalog to the internals of a query engine itself. Arguably, the suitability of Differential to on-line analytical computations would

ease some of the challenges of maintaining up-to-date metrics on relations, without incurring a significant operational overhead.

But we have also seen that 3DF's continuous query processing model breaks crucial assumptions made by traditional planners to an extent, that we do not see a heuristics based approach as viable. Summarizing the above sections, the reasons for this are manifold:

1. Cardinality estimation is simultaneously the most important and the most fragile part of query planning. Estimation errors are often proportional in the size of entire relations and compound on complex queries invoving many joins. We saw that estimation errors of many orders of magnitude are not merely a theoretical concern but a common occurrence in real-world planners.

2. In our setting, we will often have to implement computations *before* any inputs (and thus any heuristics) are available. Data sources are heterogeneous, live outside of 3DF, and will often not provide the necessary statistics. This invalidates the assumption of global-knowledge that traditional query planners rely on.

3. Even in clean, normalized data models, certain correlations between columns are near impossible to avoid in practice, because a relation is often the only feasible way of modeling discrete, functional dependencies.

4. Even assuming accurate and timely metrics, certain queries (such as the subgraph queries from section 6.4) have *no* join-at-a-time plan that materializes no more tuples than the maximum result set size.

Fortunately we have also seen that more recent approaches from the field of adaptive query processing, incremental view maintenance, and worst-case optimal join processing match our operational needs much more closely.

In the following chapters, we will now focus on what we believe to be the most significant challenges facing any system that wishes to provide continuously evaluated, complex relational queries to many interactive clients with predictable, consistently good performance. For each, we will describe the problem in general terms first, and then alongside specific examples. We will then describe and evaluate potential remedies based on the techniques we have explored in this chapter.

7 Eager Synthesis

3DFs attribute-oriented data model encourages users to think of their domain in terms of attributes, entities, and the relations that govern their interaction. Borrowing from Datalog, 3DF provides the *rule* — fundamentally a named disjunction of conjunctions — as its core abstraction to express information that can be derived from the base relations. Rules may refer to each other or recursively to themselves. Domain logic, constraints, and higher-level relationships are all expressed as rules.

It is often the case that users identify new rules that are sensible abstractions within their domain, but that when evaluated on their own materialize enormous amounts of tuples. Before this work, users of 3DF would often run into this problem when modeling a problem rule-by-rule, because 3DF would implement new rules immediately upon registration. Additionally, in order to allow users to work off of previously defined rules, 3DF would maintain rule results in a Differential arrangement. Disastrous, partially defined rules would therefore incur not just excessive time, but also space complexity.

This is a prime example of violating the robustness property identified in section 3: users are merely following the declarative interaction model, which we are actively encouraging, and get punished for doing so.

7.1 Example

An example of this problematic scenario in the context of the social graph domain is any transitive relation between persons, such as `:person/knows` or `:person/friends-with`. A common rule for recursively capturing such transitive closures might be:

```
[(knows ?a ?b) [?a :person/knows ?b]]
[(knows ?a b) [?a :person/knows ?x] (knows ?x ?b)]
```

From the users perspective, this rule is tremendously useful because it abstracts away a complex, iterative graph exploration problem for an arbitrary number of hops.

For 3DF on the other hand, immediately materializing this rule into a Differential arrangement has disastrous consequences, as it will materialize and index the full transitive closure of the social graph.

Another common instance of this problem arises when modeling access control policies. The power of the query language allows for straightforward expression of rich, property-based policies, or simple, permission-based ones such as:

```
[(access? ?user ?entity) [?user :permission/read ?entity]]
```

Eagerly deriving the `access?` relation would materialize all read permissions for all users. Even more problematic, consider a rule such as:

```
[(access? ?user ?entity) [?user :user/role "Admin"]]
```

This rule makes sense from a domain-modeling point of view, but is not even a valid rule in 3DF, because it doesn't bind the `?entity` variable. Such a rule can only ever be used in combination with additional ones.

7.2 Remedy

Luckily, generic rules are rarely used in isolation, i.e. without any more specific constraints applied (in the same sense that a function is rarely evaluated for all possible parameters, but rather for a much smaller set of specific ones).

As to the example of `:person/knows`, an application might at any given point be interested only in whether a few *specific* persons know of each other, or in exploring a limited number of hops in the social network of a person. Therefore we can remediate this problem by deferring implementation of a rule until at least one client has expressed an active interest in it.

Assume the definition of the `knows` rule is available on the server once any client has registered it, but the corresponding dataflow isn't created until another client arrives with the following query:

```
;; Is there a pair of users named Alice and Bob,
;; which know of each other?

[:find ?x ?y
 :where
 [?x :person/firstname "Alice"]
 [?y :person/firstname "Bob"]
 (knows ?x ?y)]
```

Now the resulting computation is much less problematic, because we need only explore the transitive closure between persons named either Alice or Bob. Deferring synthesis of `knows` has saved us a lot of trouble in this case. Additionally, by deferring evaluation we have obtained a complete picture of user interest. This means that we don't have to maintain an arranged version of the `knows` relation in all cases, but can immediately forward tuples downstream.

A few significant changes were necessary to change 3DF's synthesis approach from eager to lazy. These are detailed in 12.2.

7.3 Conclusions

As will be the topic of subsequent chapters, much deeper trade-offs await when deciding what queries to evaluate as a unit and what parts to maintain for reuse. Eager materialization is problematic in that we lose the ability to even reason about what the best evaluation strategy is. This makes a deferred implementation strategy less of an optimization, and more of a requirement.

We've also seen how semantically meaningful rules are often terribly underconstrained. Abstractions such as the `knows` rule increase the potential for

re-use of computations across many clients, but might dramatically increase the magnitude of the result set.

On the other hand, as is established practice in query planning, pushing more selective downstream operations back up into these generic rules can greatly improve performance, at the cost of increased *specialization* of the dataflow — and thus reduce potential opportunities for sharing it between clients. This problem is compounded by the fact that a single bad decision, when implementing a dataflow that many other flows depend on, means we are stuck with the potentially horrific plan until all downstream computations are shut down. We will explore this trade-off in greater detail in the following chapters.

8 Intermediate Join State

Within an attribute-oriented data model, relational joins cannot be treated as scary, exceptional operations, but must rather be viewed as commonplace, everyday tools. Even the most basic aggregation of data — obtaining a consistent view of domain entities over time — requires joining potentially many attributes. Robustly maintaining high-arity joins for many concurrent users with low-latency and high-throughput is therefore the defining problem of the 3DF architecture.

Join operators must maintain state to buffer inputs on either side, until matching inputs from the other side arrive (and vice versa) or the epoch closes. Systems like Kafka Streams enforce sliding, fixed-size window semantics (Inc. (2019)) to put a bound on the state maintained by each join operator. Differential Dataflow maintains such state in arrangements (c.f. chapter 4.3) that are compacted as the computation makes progress (i.e. epochs are completed). Differential's `join` operator will therefore arrange both of its input collections on the join variables, should they not already be available in arranged form.

For higher-arity joins this becomes problematic, because the output of the first two-way join will be fed as an input into the second two-way join, and so on. At each stage along the spine (except for the very last), intermediate join results will therefore be maintained in arranged form. This is illustrated in Figure 1.

Depending on the selectivity of the various join stages, this can lead to significant space overheads. Worse, in our continuous query processing setting, erroneously estimating intermediate cardinalities will therefore cause disastrous complexity in both time *and* space.

8.1 Example

Intermediate join state becomes especially apparent in star-joins along highly correlated columns, e.g. one-to-one relations with very similar cardinalities and low selectivity. Unfortunately this is a very common use case in our attribute-oriented data model. Consider the following query.

```
[:find ?person
 :where
 [?person :person/firstname _]
 [?person :person/lastname _]
 [?person :person/gender _]
 [?person :person/birthday _]
 [?person :person/creation-date _]
 [?person :person/ip _]
 [?person :person/browser _]]
```

This query requires a seven-way join on all person ids, aggregating seven different person attributes back into a consistent view of each person. Within the

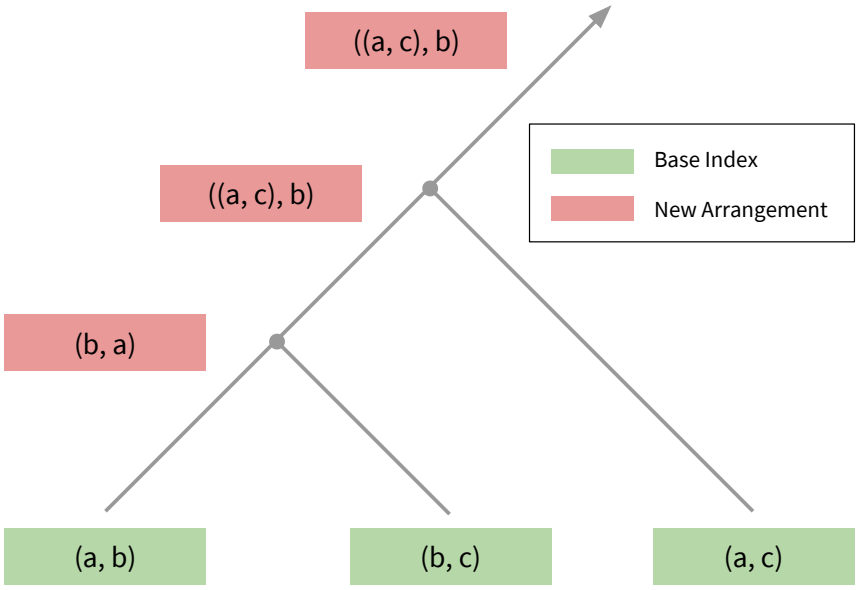


Figure 1: Join At A Time

An exemplary three-way joiin implemented via a join-at-a-time plan. Base relations are in green and are shared across all queries. Arrangements in red hold intermediate join state, are created on-the-fly, and not shared with other dataflows.

LDBC dataset (Erling et al. (2015)) for example, every person carries all seven of these attributes. Therefore all relations are of the same size and map one-to-one from person ids to the associated values. Because of this, implementing this query as six consecutive two-way joins leads to a five-fold increase in the number of tuples maintained by this dataflow *regardless of the chosen join order*.

We encounter an even more pronounced version of this problem on cyclic sub-graph queries, such as the triangle search:

```
[:find ?a ?b ?c
 :where
 [?a :person/knows ?b]
 [?b :person/knows ?c]
 [?a :person/knows ?c]]
```

Here the attributes do not have a one-to-one correspondence anymore (a few celebrities might have in-degrees in the millions, whereas most people are in mutual friendship with a few hundred others) and intermediate result sets can grow super-linearly.

8.2 Remedy

Adapting the incremental join view techniques from Blakeley et al. (1986), we can break up our queries along their potential sources of change (c.f. Figure 2). In our example, any of the comment attributes may change arbitrarily, which means we will end up with six separate queries.

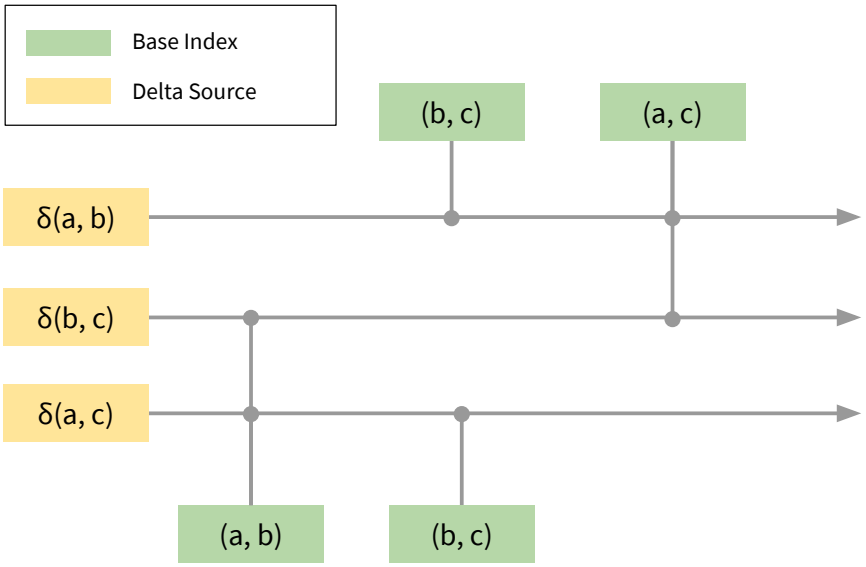


Figure 2: Delta Queries

An exemplary three-way join implemented via delta-queries. Three independent dataflows are required, each handling changes to a specific base relation. These delta sources are stateless.

What is different about the resulting plans, is that, given an indexed representation of all attributes, they can be implemented in an otherwise stateless manner. They do not need to maintain intermediate arrangements as was the case for the traditional join operator. This allows us to decouple the memory use of our join processing from the number of active queries and the complexity of the joins, and instead bound it in the number and size of attributes maintained. There is a trade-off of course, as transforming plans into delta queries leads to an increase in the number of dataflow elements.

Within any given delta flow, an attribute may only appear at a position where either of its sides is already bound by the tuples flowing through it, in order to avoid having to produce the entire relation for each incoming tuple. We therefore would want to keep attributes indexed in both directions (from entity id to value and the other way around), to increase the number of possible plans

available to us.

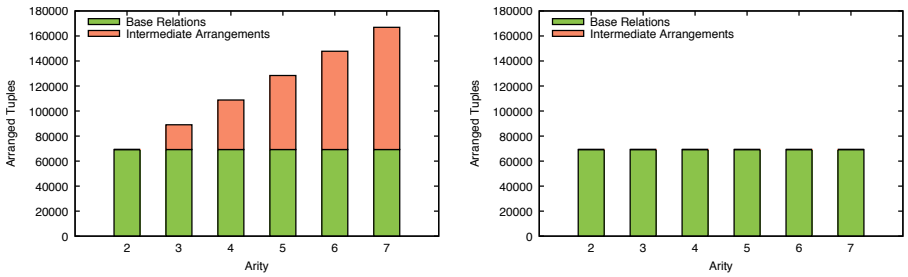
Another subtle issue, identified in McSherry (c), arises when implementing delta queries within the dataflow model. Simultaneous updates to multiple of the participating attributes can lead to redundant derivations of the same result tuple. We must be careful to impose a logical order on the execution. This implementation detail is described later in subsection 13.4.

8.3 Evaluation

We show that joins of arbitrary arity implemented with delta queries maintain no intermediate tuples, whereas the same joins implemented as join-at-a-time plans store intermediate tuples in proportion to the size of the base relations and the arity of the join. Both approaches require base relations in arranged form, which are shared across all queries. We find that delta queries do not adversely affect update latencies in a significant way.

Using the LDBC data generators we generated a dataset simulating the social network activity of 10,000 persons. On this we ran subsets of the above conjunctive query on attributes of persons, increasing the number of clauses participating in the join with each run. These runs were performed with the dataset fully loaded into 3DF and without streaming updates.

During each run we measured the number of tuples maintained across arrangements within the query dataflow. Differential provides event streams containing information about the logical size of arrangements. From those events, the number of tuples within an arrangement is easily inferred. Implementation details of the logging instrumentation are covered in subsection 12.1.



(a) Join-at-a-Time (b) Delta Queries

Figure 3: Total Number of Tuples in Arrangements

Total number of tuples maintained across base relations and intermediate join state, measured on join-at-a-time plans (a) and delta query plans (b) of increasing arity. Tuples in intermediate arrangements are not shared between queries. Intermediate state grows proportional to join arity for join-at-a-time plans, whereas no intermediate state is required for delta queries.

Figure 3 shows the total number of tuples held in arrangements, after each join produced a complete initial result set. Tuple counts are broken down into

those held in base relations, which are shared across all queries, and those stored in intermediate arrangements created by the join operator. We observe that for join-at-a-time plans, the number of arranged intermediate tuples grows linearly with the arity of the join, whereas no intermediate tuples are stored when using delta queries on joins of arbitrary arity. Note that our join-at-a-time implementation does not yet make full use of all available base relations for higher arities. An optimal implementation should be expected to produce around half the absolute number of intermediate tuples.

Arguably join-at-a-time plans allow for a lower overall memory footprint if the number of queries is low and join arities are small in comparison to the number and size of base relations, because reverse indices on base relations wouldn't have to be maintained. However these assumptions do not hold for our setting.

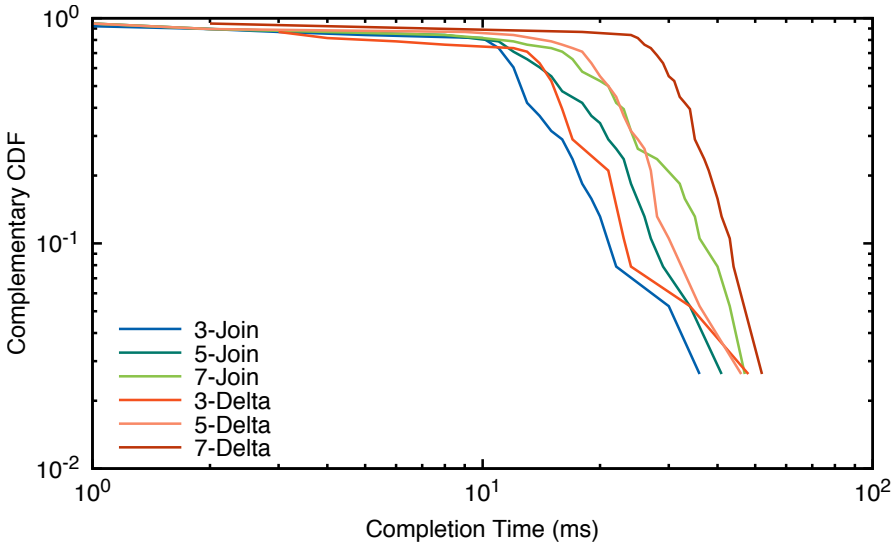


Figure 4: Latency CCDFS

Join completion times for each round of inputs, when streaming inputs into pre-registered joins of increasing arity. Delta queries do not adversely affect update latencies in a significant way.

We then inverted the setup, first creating join dataflows before streaming in all person attributes from the LDBC dataset. We measured completion times for each distinct batch of inputs. The resulting distributions are shown in Figure 4. We observe that both join strategies produce very similar distributions with increasing arity. Join-at-a-time plans achieve a slightly lower overall profile. Delta queries thus allow us to trade dataflow size in return for a predictable memory footprint, and to do so without affecting latencies in a significant way.

Considering only selections, projections, and n-way joins, the overall system could be configured to use only memory proportional to the number and size of

the base relations. No additional operator state is required, independent of the number of queries and the arity of the joins within them.

Additionally, delta queries decouple space from time complexity. Using traditional tree joins, a sub-optimal join plan incurs both a heavy computational cost as well as space overheads, due to the sheer number of intermediate tuples materialized and maintained in arrangements. Delta queries still have to spend time processing large numbers of tuples, but without the need to index them at intermediate stages.

While we must at some point take measures to keep the overall size of the dataflow manageable (as will be the topic of section 10), we conclude that delta queries are a powerful technique in many common scenarios.

9 Sub-Optimal Join Ordering

As has been widely documented within the database community (Leis et al. (2015), Lohman (2014)), many complex queries are highly sensitive to join ordering. We have discussed the established heuristical approaches originating in section 6 and hinted at their shortcomings when applied within the continuous query processing setting. There we concluded that in practice, cardinality estimation under the traditional assumptions (uniformity, uncorrelated join predicates, overlapping key domains, and full access to meaningful statistics) can not effectively avoid disastrous plans on long-lived query plans.

We would like to take an adaptive, defensive approach that makes disastrous plans impossible, even if at the expense of best- and average-case performance. The reasons for this are two-fold: first and most importantly, disastrous query plans violate all three of the desired properties outlined in section 3. Second, from our experiences with 3DF itself, and from experiences with other declarative systems such as the Prolog language, we know that the mere *possibility* of unpredictable, severe performance degradations breaks the fundamental promise of the declarative abstraction, and force users to reason defensively about the system runtime.

9.1 Example

We can distinguish between different classes of problematic queries:

- **Worst-Case** Queries for which any join-at-a-time plan will process asymptotically more tuples than could possibly be contained in the result set. A class of queries for which this is case is formalized in Ngo et al. (2012). Simplifying somewhat, these are queries that compute the d-way join of all $(d-1)$-ary projections of the result set. An instance of this query class is the triangle query, which computes (a, b, c) from its projections (a, b), (b, c), and (a, c). Such a query is called *cyclic*, because its hypergraph representation (with variables as nodes and relations as edges) contains cycles.

  Triangles and cyclic sub-pattern queries in general have many applications. The LDBC Social Network Benchmark query `BI/read/11` for example, contains a triangle between comments, messages, and tags, in order to find unrelated replies. Query `BI/read/19` asks for a 4-clique between persons, comments, and messages, in order to determine for each person the set of strangers they have interacted with.

- **Average-Case** Queries across correlated columns, i.e. on data where the assumptions of uniformity, independence, and overlapping domains are violated to varying extents. For queries in this class, accurate estimation of cardinalties can make a huge difference for join-at-a-time plans. These are queries whose hypergraph representation is cycle-free.

LDBC query `BI/read/2` for example, joins countries, cities, persons, messages, and tags. The order of this traversal can make a significant difference, depending on the relative selectivity, and additional predicates applied to individual relations (e.g. "consider only cities in germany").

- **Best-Case** Queries across uncorrelated, uniformly distributed columns. Their hypergraph representation is cycle-free, and the assumptions of traditional cardinality estimation hold.

  Examples would be star-joins on uncorrelated columns, where each column holds at most a fixed number of values for each key. E.g. joining user ids exclusively on single-cardinality attributes such as username, email, or age.

9.2 Remedy

We covered recent advances in the area of worst-case optimal (*wco*) join processing that describe n-way join algorithms with improved asymptotic complexity for queries in the worst-case class. These algorithms distinguish themselves from traditional approaches by checking cardinalities for each individual input tuple at processing time. They can ensure that any extension of a set of prefixes to a new variable will look at the smallest number of tuples, even in the face of correlated or non-uniformly distributed relations. Of particular interest to our setting is the *Delta-BiGJoin* family of algorithms due to Ammar et al. (2018), which are designed for the distributed, data-parallel dataflow setting. Delta-BiGJoin utilizes the same delta query technique covered in the previous chapter.

An exemplary delta query extending changes on a relation (a, b) to valid (a, b, c) tuples is illustrated in Figure 5. The extension is split across three steps, *count*, *propose*, and *validate*. Each step is reminiscent of adaptive operators such as eddies, in that it dynamically routes inputs between multiple candidate relations. To be considered a candidate, a relation must bind the target variable (c in this case) and one of the prefix variables (a or b).

During counting the input tuple is routed through all candidate relations, s.t. they may provide the number of extensions that they *would* propose, would they be asked to do so. Accordingly, during propose, the input tuple is routed to the candidate that claimed the fewest extensions in the count step. Finally, in validate, we intersect the winning candidate's proposals with those of all other candidates, leaving us with all valid (a, b, c) tuples for that specific prefix. These steps are performed for each input tuple and ensure that no more tuples are materialized than could possibly be part of the result set.

By composing this extension primitive, this approach can be generalized to joins of arbitrary arity. By taking the union across separate count-propose-validate delta queries we obtain again an incrementalized operator that responds to changesin *any* of its inputs. Finally, Delta-BiGJoin can be generalized to maintain queries involving predicates, stratified negation, and other types of

constraints. An in-depth explanation of our implementation and these extensions is given in the chapter on implementation details, section 13.

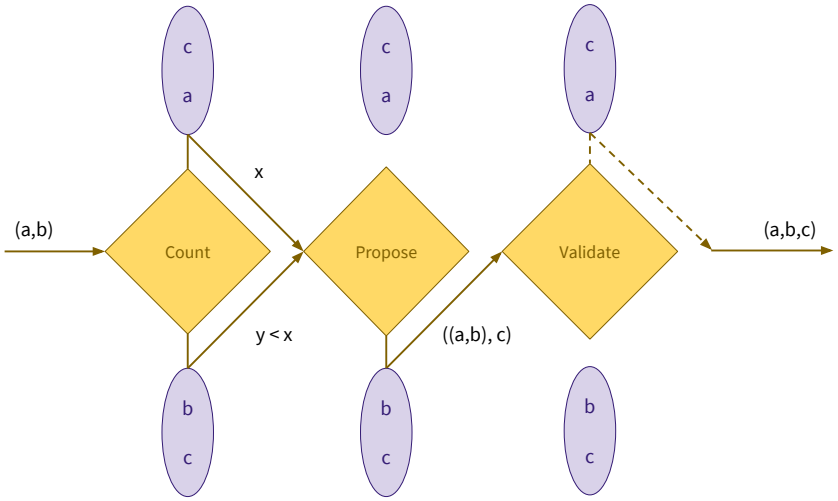


Figure 5: Worst-Case Optimal Dataflow Join Primitive

This operator reacts to changes in a single base relation (a, b) and maintains a collection of valid (a, b, c) as output, corresponding to the result set of joining $(a, b), (b, c), (a, b)$. In doing so it materializes no more tuples than could possibly be contained in the output.

Delta-BiGJoin is robust against adversarially skewed datasets, adapts as distributions change over time, and can improve the asymptotic complexity of computing cyclic queries. For many average- and best-case queries however, suboptimal join orderings have an adverse effect on performance that is significant in practice, although not asymptotically worse. In these cases the variable order chosen by a wco join matters much more.

For example, given prefixes (a, b) and relations $\{r_1(a, c), r_2(a, d)\}$, we might choose to first extend to c via r_1, or instead to d via r_2. Choosing one over the other has no bearing on the asymptotic result set bound (Ngo et al. (2012)). Still, for non-uniformly distributed relations, some prefixes might produce many more matches in r_1, whereas for others it would be the other way around. Picking the optimal variable order is thus again a data-dependent cardinality estimation problem, if one with a better safety net. We discuss this problem later on in section 13 but a satisfying solution is beyond the scope of this work.

Further, as seen above, wco algorithms perform significantly more work per tuple than traditional two-way joins. For example, at each extension stage, each tuple requires multiple index lookups in order to determine the optimal

extension strategy.

We thus expect a wco approach to significantly improve robustness on queries in the worst-case class, but to adversly affect performance on average- and best-case queries, as compared to traditional, heuristics-based approaches. What we gain is adaptive, continuous maintenance without the need for synchronized re-optimization steps, and compatibility with the memory-efficient delta query strategy from section 8.

9.3 Evaluation

We show that our implementation of Delta-BiGJoin leads to significantly more predictable latencies on a representative cyclic query, outperforming sub-optimal join-at-a-time plans by several orders of magnitude. Conversely, we show that the performance of different join-at-a-time strategies varies dramatically with the chosen join order. While the best join-at-a-time order matches the performance of our worst-case optimal approach, the worst one leads to an unresponsive system that quickly runs out of memory. We also show that our worst-case optimal approach can match the performance of join-at-a-time plans on queries in the average-case class, while adding roughly an order of magnitude overhead to queries in the best-case class.

As a representative cyclic query, we chose the triangle query $(a, b), (b, c), (a, c)$. We ran this query on the livejournal graph dataset, which contains 68 million edges. We introduced edges in node order, feeding them into a single base relation. The i-th round of input is therefore that in which all outgoing edges of node i are ingested. We used a single worker, running on a single core of a 2.7 GHz Intel Core i5, with 16GB RAM available. We ran this setup for both strategies, join-at-a-time and worst-case optimal, and for three different input clause orderings each, measuring completion time for each round of inputs. The resulting latency distributions are shown in figure 6.

Due to the worst join-at-a-time strategy running out of memory on iteration 87, we ran an extended setup without it. The extended setup was run on the same dataset, on a single core of a 3.4 GHz Intel Core i7 with 64GB RAM available. We also increased the batching from one to 1000, ingesting the outgoing edges of 1000 nodes in each round of inputs. The resulting distributions are shown in Figure 7.

We observe that the join-at-a-time strategy is highly sensitive to clause order. In particular, the worst ordering $(((a, b) \bowtie (a, c)) \bowtie (b, c))$, never completes inputs at node 87 before running out of memory, while the best ordering $(((a, c) \bowtie (b, c)) \bowtie (a, b))$ comes very close to the wco performance. The wco strategy performs consistently well, regardless of clause order, and consistently outperforms the two disastrous join orderings by one to many orders of magnitude. On single node batches, the best join-at-a-time plan matches the performance of the worst-case optimal strategy. On the extended setup we find the performance of both non-disastrous join-at-a-time plans to converge. The worst-case optimal strategy outperforms both by an order of magnitude.

Next we evaluated performance on the `BI/read/2` query from the LDBC

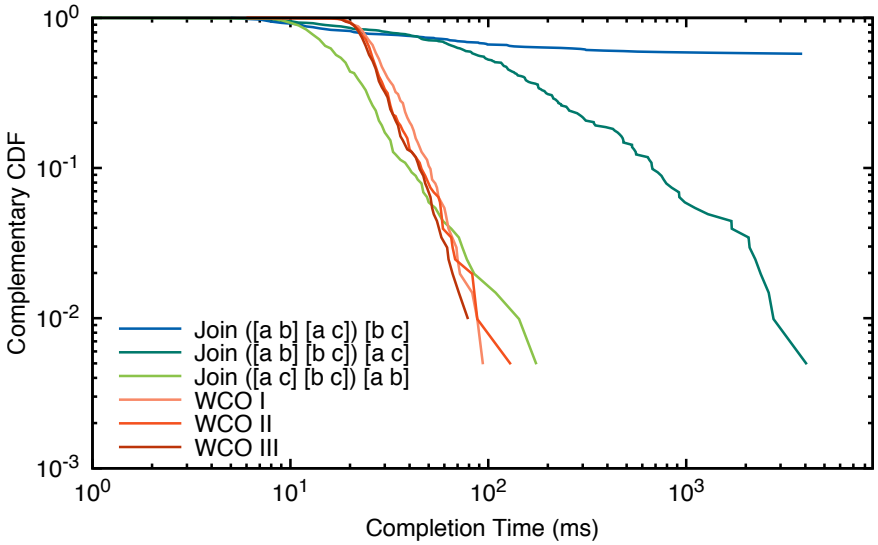


Figure 6: Triangle Query

Join completion times for each round of inputs, while maintaining a triangle count over a stream of the livejournal graph. The worst-case optimal strategies perform consistently well, independent of the provided clause order. They match the performance of the best-performing join-at-a-time plan, while outperforming sub-optimal orderings by several orders of magnitude.

benchmark, running with a single worker and on a 2.7 GHz Intel Core i5, with 16GB RAM available. We used the LDBC data generator to generate a dataset simulating the activity of 10,000 users in a social network. The resulting dataset is a gigabyte in size and translates into roughly 40 million tuples when normalized. We registered each query plan and then ingested the data in small batches of about 10,000 tuples at a time, measuring the completion time for each input epoch. Figure 8a shows the resulting latency distributions for two different join-at-a-time orders and the worst-case optimal strategy. In this case, performance of different join orders is distinguishable but very similar. The worst-case optimal strategy matches the performance of both join-at-a-time plans.

Finally we evaluated a simple star-join of posts across four one-to-one attributes (`:post/ip, :post/browser, :post/language, :post/content`). The dataset contains about a million posts, each having exactly one value assigned for each of these attributes. All data was ingested using the same setup as for the average-case query. Figure 8b shows the resulting latency distributions. Here the worst-case optimal strategy adds an order of magnitude in overheads compared to the join-at-a-time plans, both of which perform essentially identical.

While more extensive evaluation across a larger set of queries is necessary before drawing broad conclusions, we believe that worst-case optimal join al-

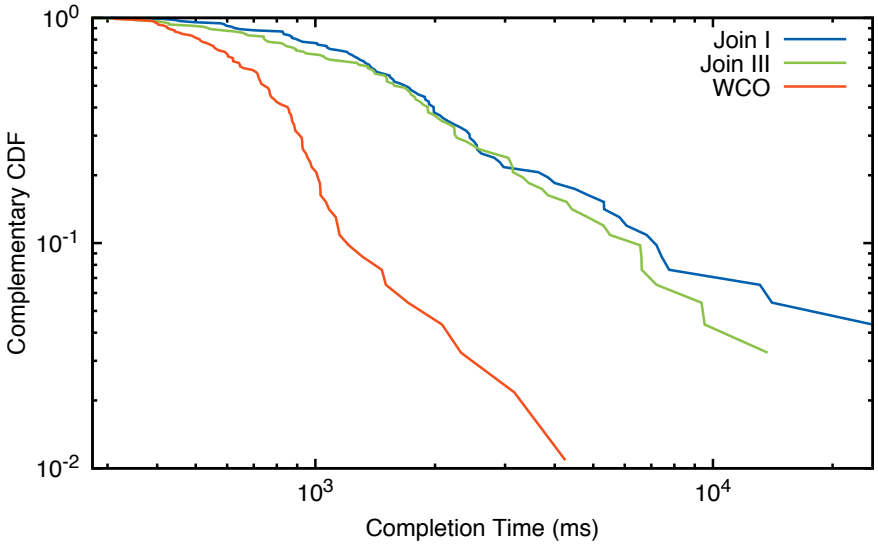
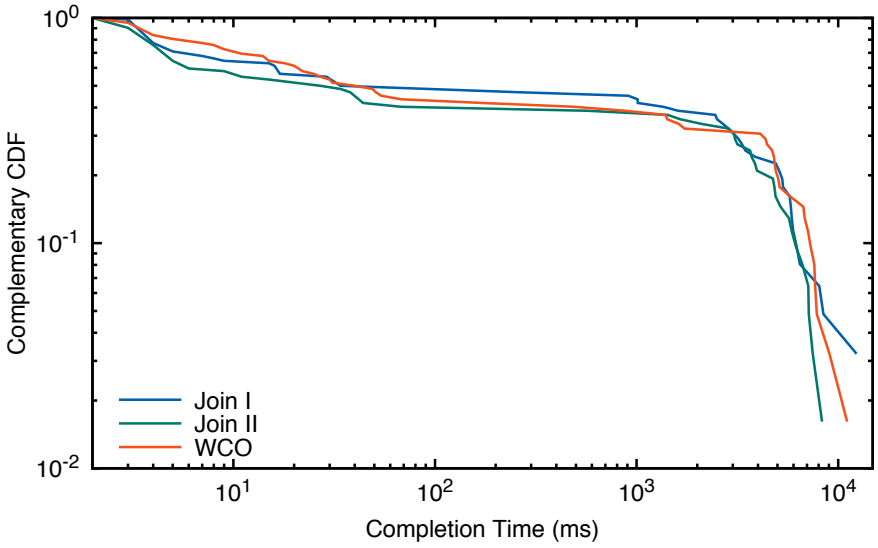


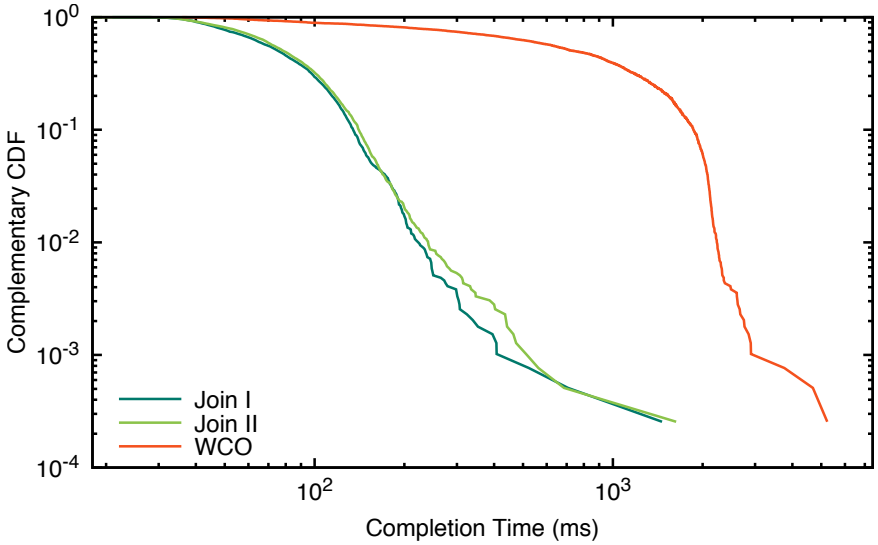
Figure 7: Triangle Query - Extended
Join completion times while maintaining a triangle count over a stream of the livejournal graph, with increased batching. The worst-case optimal strategy consistently outperforms both join-at-a-time strategies.

gorithms can achieve more predictable performance on cyclic queries whithout dramatically affecting absolute performance on simple, cycle-free ones, and without relying on brittle heuristics. Additionally, the Delta-BiGJoin family of joins is inherently compatible (and even relies on) the delta query technique discussed in the previous chapter.



(a) BI/read/2

Join completion times while maintaining the LDBC BI/read/2 query over a stream simulating the activity of 10,000 users in a social network. The worst-case optimal strategy matches the performance of join-at-a-time strategies.



(b) Best-Case Star-Join

Join completion times while maintaining a four-way star-join across one-to-one attributes over a stream simulating the activity of 10,000 users in a social network. Join-at-a-time plans outperform the worst-case optimal strategy by an order of magnitude.

10 Redundant Dataflows

Throughout the previous chapters we have seen how to continuously maintain high-arity joins with robust, adaptive performance, and within a constant memory footprint, using delta queries and worst-case optimal join algorithms. While these measures had a slight negative impact on absolute query performance in the best-case, they enabled asymptotic improvements for many other common scenarios and in any case maintained a near real-time characteristic. Still we have paid for these optimizations at the expense of redundancies in the overall dataflow. Apart from the attribute indices, we have ignored almost all opportunities for sharing resources between dataflows.

Query dataflows consume a number of resources over their entire lifetime, which might be measured in minutes or hours for interactive sessions, but in days, weeks, months, or even years for dataflows between backend services. The primary resource consumed by a dataflow element is space to buffer and index its inputs. We refer to this as *operator state*. To a lesser extent dataflow elements themselves incur some marginal costs both in space, for maintaining a representation of the dataflow graph, and in time, because additional operators need to receive and process progress information. Naturally, sharing neither indexed operator state, nor dataflow elements, limits the number of clients we can support concurrently.

10.1 Example

Consider a scenario of many analysts concurrently interacting with the same data streams. Many of them will be interested in a common subset of data, e.g. users that are currently active in Europe, but will impose some additional constraints depending on their respective tasks at hand. A more concrete example might be the comment stream from previous chapters:

```
[(comment ?comment ?place ?person ?ip ?content ...)

 [?comment :comment/creation-date ?creationDate]
 [?comment :comment/creator ?person]
 [?comment :comment/ip ?ip]
 [?comment :comment/browser ?browser]
 [?comment :comment/content ?content]
 [?comment :comment/place ?place]]
```

We have multiple analysts, each interested only in comments from a specific location. For an analyst interested in comments originating in Zurich, a query might look like the following:

```
[:find ...
 :where
 (comment ?comment ?place ...)
 [?place :place/name "Zurich"]]
```

A similar situation naturally arises when implementing data access policies, which force clients to only work with certain subsets of all available data. Datalog makes it possible to succintly express even complex, property-based policies. A set of rules describing a simplified, role-based access control scheme might look like:

```
[:find ?document ?content
 :where
 (access? <client-id> ?document)
 [?document :document/content ?content]]

[[(access? ?client ?document)
  [?client :client/role ?role]
  (role-access? ?role ?document)]

 [(role-access? ?role ?document)
  [?role :role/access? ?document]]

 [(role-access? ?role ?document)
  [?child :role/super ?role]
  (role-access? ?child ?document)]]
```

Intuitively we might point out that recursively re-deriving the `role-access?` relation for each client seems wasteful.

10.2 Sharing Dataflows

In general, given n analysts, we can service their queries in a few different ways.

Shared Nothing We create n indepedent dataflows, sharing only the base relation indices. This gives us the greatest freedom to pick the best possible plans for each individual analyst, for example making use of the worst-case optimal join framework covered in chapters 6.4 and 9. On the other hand, all of those n dataflows will perform a lot of redundant work and incur significant overheads, deriving n copies of the `comment` relation.

Shared Trunk Differential's *arrangements* allow us to re-use materialized relations between dataflows. We could therefore create a single dataflow deriving the `comment` relation, on top of which n specialized dataflows will be only be responsible for implementing the specific additional joins requested by each analyst. Because arrangements materialize tuples into sorted batches, doing so will cost extra storage proportional in the size of the `comment` relation, on top of the storage taken up by base relations (c.f. 8).

Sharing a common part of the dataflow (a *shared trunk*) avoids redundant derivations of the `comment` relation and significantly reduces the total number of dataflow elements created (a super-linear reduction in case of delta-queries), at the cost of some extra storage — seemingly a fine trade-off!

On the other hand, materializing a shared trunk fixes the space of its possible implementations. In particular (as touched on in 7) it prevents us from

exploiting highly selective constraints provided by individual analysts to cut down on the number of tuples produced by each flow. In the specific case of the `comment` relation, this might not be a problem, as the number of tuples produced during its derivation is proportional to the number of comments in the system. For other relations (for example the transitive graph induced by the `:person/knows` relation) it might lead to catastrophic plans.

Whenever we are considering to build multiple dataflows off of a shared trunk, we must therefore estimate whether the shared trunk is *sufficiently selective*. Increased specialization opens optimization potential, whereas increased generalization opens potential for resource re-use. After all our work to avoid heuristics and cardinality estimation, we would not want to reintroduce this element of unpredictability into the behaviour of our system. Computing multiple redundant instance of a plan where worst-case optimal execution improves the asymptotic complexity can still be significantly cheaper than sharing the sub-optimal plan. Additionally, once we've made the decision to share a flow, we are commited to maintaining it for as long as any dependent flows are active.

Consistent with our robustness goal we therefore want to err on the side of caution at the expense of reduced best-case scalability.

10.3 Multi-Tenant Dataflows

In some situations, an additional implementation strategy might be available to us: we can model multi-tenancy within the data plane itself. Recall the query from earlier:

```
[:find ...
 :where
 (comment ?comment ?place ...)
 [?place :place/name "Zurich"]]
```

As a first step, we will re-write this query, replacing the explicit filter by a parameter input, drawn from an additional parameter attribute.

```
[:find ...
 :where
 (comment ?comment ?place ...)
 [?place :place/name ?name-filter]
 [_ :parameter/place-name ?name-filter]]
```

This is useful in an of itself, because it gives us the ability to update the filter parameter (with updated results derived incrementally as usual), without registering separate queries. Once a query is parameterized in this way, we can allow multiple concurrent users to interact with the same parameter collection. Of course we must then make sure to isolate users from each other and route results correctly.

In this (admittedly simple) situation we can use a single dataflow for all n analysts, while still being able to move the `:place/name` constraints into

the same, worst-case optimal query plan. Still, real-world use cases (such as the access control policy from above) can already benefit from multi-tenant parameterization, via the following query:

```
[:find ?tenant ?client ?document
 :where
 (access? ?client ?document)
 [?tenant :parameter/document ?document]
 [?tenant :parameter/client ?client]]
```

The resulting dataflow can be used to compute subsets of the `access?` relation for multiple concurrent tenants, and for those subsets to be redefined dynamically at runtime.

We find precedence for this approach in the NiagaraCQ system by Chen et al. (2000).

> NiagaraCQ implements sharing even when operators differn in certain constants, by implementing the operators using relational joins against the table of constants Chen et al. (2000).
>
> Hirzel et al. (2014)

It is also reminiscent of more recent work on partially-stateful, multi-tenant dataflow systems by Gjengset et al. (2018).

10.4 Identifying Opportunities for Reuse

Of course, reuse via parameterized, multi-tenant dataflows is not possible if users request different conjunctions. Finding common sub-structures across query plans that would be suitable for either of the approaches outlined above is in general a very computationally intensive problem and beyond the scope of our inquiry.

Declarative languages allow for simplified, pragmatic approach. Instead of arbitrary re-use across all clauses in all rules, we only consider re-use at rule boundaries. To facilitate this, each rule should be given a unique name. We can further break this down and give unique names to all conjunctions, e.g. by creating separate rules for the different branches of an `or` clause automatically. Additionally, we can split off plan stages that do not affect the underlying joins, such as projections, transformations, and aggregations. At the time of this writing, this is not done automatically by 3DF.

10.5 Reuse And Worst-Case Optimal Dataflow Joins

The worst-case optimal join operator outlined in chapter 9 warrants special consideration w.r.t. resource sharing.

First, most such algorithms (and in particular our implementation, c.f. chapter 13) assume that participating relations are available, indexed by each possible subset of their bound symbols. For a binary relation (e, v) two indices would

have to be maintained (one from e to v, and one reverse index). For a ternary relation, e.g. (a, b, c), already six indices are required.

We therefore, by default, break derived n-ary relations down into their base relations when using them from within another query, rather than share the already derived relation via Differential's arrangements. Re-using n-ary relations via a Differential arrangement would force us to create the additional required indices on the fly. On the other hand, this means that relations are rederived across a potentially large number of dependent queries.

However, leaving aside the considerations from above on whether we should reuse, reuse of derived binary relations (e.g. `(access? ?user ?document)`) is possible via the same indexing structures used for base relations. Within the dataflow model, index arrangements do not need to distinguish whether they are being fed by external inputs or from a derived relation.

10.6 Other Strategies

Again from Hirzel et al. (2014) we learn that

> YFilter implements sharing between applications written in a subset of XPath by compiling them all into a combined NFA [Diao et al. 2002].

Similar NFA-based approaches can be found in the literature on complex event processing (e.g. Agrawal et al. (2008)). While such methods are available to us in Timely Dataflow (e.g. via the built-in `StateMachine` operator), they do not fit cleanly into the relational model implemented by 3DF and Differential Dataflow. The question of whether the two approaches can be combined in a sensible way is left for future inquiry.

10.7 Evaluation

We show that sharing dataflow resources via arrangements significantly improves the scalability compared to a shared-nothing approach, but is still limited by the number of dataflow elements. Sharing a single dataflow between multiple tenants, with isolation handled in the data plane, is applicable only in specific scenarios, but scales dramatically better.

For this experiment we generated a constant load of 500 tuples per second, simulating sensor measurements reported by different devices. Every second of the experiment, ten new clients were introduced using one of three strategies: (a) shared-nothing, (b) sharing via arrangements, and (c) multi-tenancy in the data plane. Input epochs close every second, we measure the latency to close every epoch. The results are shown in Figure 9. We observe that data plane multi-tenancy effectively uncouples scalability from the limits imposed by overall dataflow size.

While promising, these observations leave a lot to be desired. In particular, the behaviour across a larger set of queries, the applicability of multi-tenant

sharing, and scalability on a more realistic, distributed workload requires significant further investigation. Due to the immaturity of 3DFs networking code, meaningful simulation of such a setup was not possible within the scope of this work, but will be a topic of ongoing interest for us.

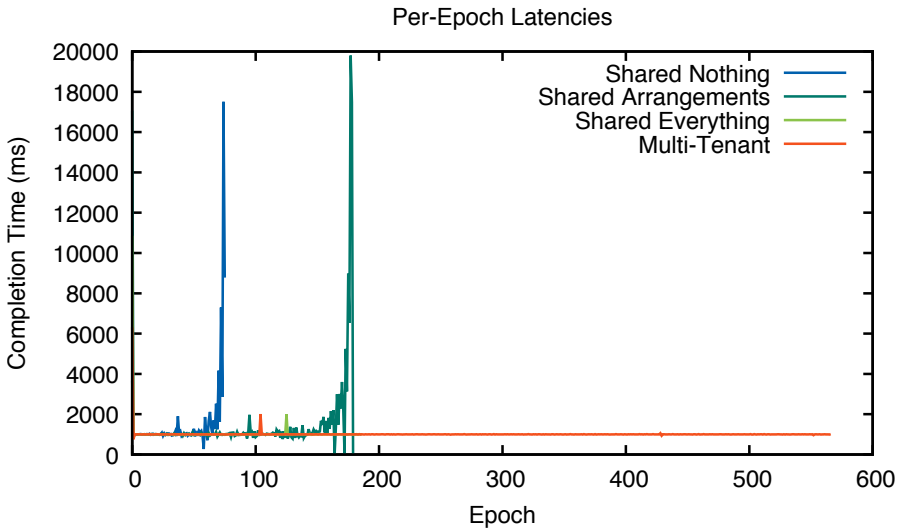


Figure 9: Single Worker

Different resource-sharing strategies under a constant load of 500 tuples and ten new clients every second. Overall dataflow size limits scalability (shared-nothing). Sharing resources via arrangements improves scalability significantly. Multi-tenant dataflows share all dataflow elements and isolate clients in the data plane. This strategy is therefore unaffected by limits on the dataflow size.

11 Skew

The techniques discussed throughout the previous chapters, when taken together, allow us to treat many-way relational joins as safe, efficient operations in dynamic, multi-user environments. In the context of 3DF specifically, they enable flexible data modeling that can evolve with the needs of an organization and provide the foundation on which the entire expressive range of a language like Datalog is implemented. While continuously changing data distributions posed a challenge to traditional methods of cardinality estimation, skewed datasets cause additional problems in a distributed setting.

Skew problems arise whenever inputs are not distributed evenly across workers. For stateless operators such as `map` or `filter`, we are free to balance inputs evenly amongst workers. This is generally not possible for stateful operators. In particular, an operator such as `join` must be certain it will see all inputs for a key k on a specific worker, if it has seen any input for k on that worker. The physical distribution of data will therefore tend to follow its logical distribution — which is rarely uniform in practice.

11.1 Example

An extreme example of skew is caused by reverse indices (mapping attribute values to their keys) on categorical data. Our worst-case optimal dataflow join implementation relies heavily on reverse indices on attributes (this is discussed in detail later in section 12). Categorical data is the cornerstone of many modern analytical tasks. Predictive models both consume and serve as natural input sources of categorical inputs (*features*) such as `:transaction/fraudulent?` or `:person/bot?`.

While features can be modelled as boolean attributes, useful features should be highly discriminating and thus apply to very small subsets of entities with little overlap. Explicitely storing false values to indicate absence of a feature is therefore a very storage-inefficient approach. For this reason, a value of true is often meant to indicate the presence of the feature on the given entity, while any entity not contained in the relation is assumed to not posses the feature. Modeling categorical features via binary relations in this way leads to an extremely skewed distribution of values in the reverse indices, which would then be stored entirely on a single worker.

11.2 Remedies

Skew in this scenario is caused by the requirement that arranged state be distributed according to its keys. To remediate this, we must modify operator implementations to either still produce correct outputs when individual workers might not see all inputs on a specific key or obviate the need for the skewed indices entirely.

Ammar et al. (2018) propose a skew resilient version of the *Delta-BiGJoin* algorithm. Its implementation and adaption to 3DF are beyond the scope of

this work.

Another possibility is to distribute one side of a two-way join evenly and broadcast all inputs on the other side to all workers. This is advantageous in situations where the volume of inputs is low on one side of the join. However, this approach does not rid us of the sub-optimal modeling of features.

Instead, we can extend our worst-case optimal join framework slightly, to cover unary relations. Unary relations bind only a single variable and thus don't require reverse indices. How such extensions work is covered in section 13. For this specific one we only provide a sketch here, as at the time of this writing, we have not implemented it in 3DF.

We observe that for unary relations, the count, propose, and validate steps of the worst-case optimal dataflow join can be serviced by a single index maintaining the entity ids in the relation in sorted order. Count, propose, and validate then all correspond to lookups in that index. The implementation is otherwise analogous to that for attributes, as described in section 13.

12 General Implementation Details

In this chapter and the following, we will cover in greater detail the changes and extensions to 3DF that we implemented over the course of this work. This chapter covers logging and other instrumentation that was added to perform the experiments, the move from eager to deferred synthesis, and support for multi-tenant dataflows, whereas the next chapter is dedicated to the implementation of delta queries and our worst-case optimal dataflow join operator.

12.1 Logging and Instrumentation

A fair bit of instrumentation was necessary in order to monitor the dataflow graph as it evolves over time, the number of tuples held in arrangements, and many other metrics. We built dataflow sources converting Timely's internal logging streams into higher-level log attributes and analyzed them interactively at run-time, using 3DF itself.

Timely Dataflow provides an extensible set of internal dataflow streams, onto which internal events are published. Logged events include among others the creation, shutdown, and scheduling of operators, activity on communication channels, and progress tracking. Differential Dataflow uses the same subsystem to provide additional information about the creation and merging of trace batches. These are sufficient to infer, for any relevant time, the number of arranged tuples maintained at each arrangement in the dataflow.

3DF attributes can be fed by arbitrary data sources. This is facilitated by *source* operators, which read from external data sources and convert inputs to one or more Timely streams. We created two new 3DF sources, `TimelyLogging` and `DifferentialLogging`.

The `TimelyLogging` source allows clients to source only the subset of logging attributes that they are interested in. It also performs some pre-processing on the raw logging streams, converting hierarchical scope addresses into the corresponding edges of the dataflow graph.

Similarly, the `DifferentialLogging` source will not expose batching and merging events directly, but rather derive from them changes to the number of tuples held by each arrangement. Batch events imply an increase in the number of tuples held (by the size of the batch), merge events imply a decrease (the difference between the compacted size of the merge result and the sum of the size of the input batches).

Taken together, this makes it easy to write queries such as the following, asking for the total number of tuples arranged at operators that haven't been shut down.

```
;; total number of tuples arranged at alive operators

[:find ?name (sum ?size)
 :where
 [?x :differential.event/size ?size]
```

```
[?x :timely.event.operates/name ?name]
(not [?x :timely.event.operates/shutdown? true])]
```

For making accurate measurements it also became necessary to side-step 3DFs default behaviour of forwarding results directly back to clients, as doing so would introduce serialization and network latency. We therefore introduced a new type of plan stages called *sinks*, as well as a sink implementation for handling our measurements. This sink swallows all its inputs and keeps track of the computational frontier across all workers. Whenever the frontier advances (thus indicating the completion of a round of inputs), it will append the number of milliseconds since the close of the previous epoch to a logfile. It will also report whenever the frontier becomes empty, indicating the completion of the entire computation.

We then extended the 3DF API in order to allow clients to specify an optional sink configuration that, when provided, will cause 3DF to forgo forwarding results via WebSocket and instead process them with the specified sink. Additionally, we extended the API to allow for an optional granularity (in seconds), indicating the maximum rate at which a client desires outputs. A granularity of one, for example, would cause outputs to be delayed and consolidated up to the next full second. This made it easy to monitor e.g. tuple counts on the live dataflow. Finally, we added an option to disable logging on queries, such as not to run into infinte loops when logging queries that consume the logging relations themselves.

12.2 Lazy Synthesis

In section 7 we have seen that demand-driven synthesis of query plans is a pre-requisite to avoid needlessly computing rules which might materialize large numbers of tuples, but which in practice would never be used without other constraints of higher selectivity present. Eager synthesis of queries limits the available plan space drastically. We changed 3DFs implementation strategy from eager to lazy, introducing a mechanism to track plan dependencies in the process.

First, we split the existing query registration API into a two-step process: *registration* and *interest*. We extended 3DF worker state with a rule store. Upon receiving a registration request, workers now merely store the provided plan (assuming it doesn't clash with a known rule of the same name). A rule is only synthesised once the first interest request depending on it is received.

Therefore, we added a mechanism to gather plan dependencies. Queries can depend on attributes and on other rules, that were registered in previous requests (maybe by other users). Like all other aspects of synthesis, dependencies are collected recursively, starting at the leafs of a query plan. The resulting set of rules requiring synthesis is brought into a canonical order, to ensure that the same exact dataflow graph is created on all workers. This is necessary, as Timely creates operator and channel identifiers based on their creation order. Redundant dependencies are removed.

At this point a decision can be made for each named dependency, for which an existing arrangement is available, on whether to reuse the arrangement or resolve the dependency into its bindings and re-synthesize them from scratch. For the reasons explored in chapter 10, the default policy is to always synthesize from scratch, reusing only attribute indices.

With these changes, clients can define complex queries interactively, across many different rules, without bringing the system to a halt. Storing plan representations has other benefits, which we might explore in future work. In particular, such a representation could be used to dynamically query the implied graph of rule dependencies and correlate it with the synthesized dataflow graph, whose structure can be derived from the logging stream, as explained in the previous section.

12.3 Multi-tenant Routing

In section 10 we saw that multi-tenant dataflows, where applicable, allow us to serve many clients without creating additional dataflow elements, and without limiting the optimization potential of the overall dataflow. We added a new routing strategy, in order to support such multi-tenant dataflows in 3DF. Our current implementation is not automated, in that it requires clients to indicate which dataflows they would like to share.

In order to be dynamically parameterizable at runtime, a query's fixed parameters must first be replaced by joins on special parameter relations. E.g.

```
;; static
[:find ?speed
 :where [<x> :device/speed ?speed]]

;; dynamic
[:find ?device ?speed
 :where
 [?device :device/speed ?speed]
 [_ :param/device ?device]]
```

We achieve multi-tenancy by tagging inputs on parameter relations with a token, uniquely identifying a tenant. Client tokens are also included as part of the output projection. Tenants register their interest, indicating the offset at which tenant tokens are to be found within the output tuples.

```
[:find ?tenant ?device ?speed
 :where
 [?device :device/speed ?speed]
 [?tenant :param/device ?device]]
```

What remains is the correct routing of results back to clients, according to the tenant token in each output tuple. This way, a given tenant only sees

the results that are of interest to them. Our current implementation re-uses the connection tokens assigned by 3DF's WebSocket server. Client tokens are unsigned 64-bit integers.

Timely Dataflow makes use of *exchange pacts* to define how data is routed between operators. For single-tenant dataflows, all workers therefore must be aware of which of them maintains the client connection. For a given request, we call this worker the *owner*. Dataflow results are then exchanged using a static policy (w.r.t to the dataflow's data plane), which routes every output tuple to the owner. Each worker additionally maintains a mapping of query names to the client tokens, which have expressed interest in receiving results from that query. As long as all relevant results are exchanged correctly to the owning worker, all interested clients will therefore receive copies.

Knowledge about the owner is broadcasted during *sequencing*, a synchronizing dataflow into which all workers input all requests received from clients. Sequencing ensures, that all workers process the sequence of commands in the exact same order. Workers tag requests that they have received from a client with their own worker id, before broadcasting them via the sequencing dataflow.

In the multi-tenant setting, the routing policy is no longer static, but rather data dependent. Workers must therefore maintain a mapping of tenants to their owning workers. New entries into this mapping are created on interest requests, and removed whenever clients disconnect or issue a uninterest request. Owning workers broadcast notifiations, whenever any of their clients disconnects, s.t. all other workers may clean up their mappings.

13 A Worst-Case Optimal Join Operator

For chapters 8 and 9 we extended 3DF with a n-way operator based on the Delta-BiGJoin implementation by Ammar et al. (2018). Delta-BiGJoin itself relies on the delta query technique, allowing us to evaluate both delta queries, as well as worst-case optimal join strategies. The present implementation is based off of McSherry (c) and McSherry (a), which are implementations of Delta-BiGJoin for generalized motif-tracking.

Over the following sections we describe the basic operator implementation and detail the changes that were required to generalize it to 3DF's hypergraph data model and indexing structures. We also added additional implementations of the core interfaces, in order to support constant bindings, binary predicates, and stratified negation. We discuss suitable variable orderings, and the use of logical timestamps to avoid redundant derivation of results across multiple delta queries. The resulting operator is called *Hector* and will be referred to as such throughout this chapter.

13.1 Basic Operation

At a high-level, the Hector operator provides the following capability: *Given a set of clauses and a set of target variables, find all possible variable assignments that satisfy all clauses.* Clauses represent the different kinds of *variable bindings*. The clause `[?e :edge ?v]`, for example, constrains variables e and v to elements of the `:edge` relation, whereas the clause `[(< ?v 1000)]` restricts the variable v to values less than 1000.

Some bindings, such as predicates and negation, are exclusively constraining (shrinking the space of possible values for a given variable), others, such as attributes and constants, can also provide values. We will refer to the former as constraints, and to the latter as bindings from now on.

If no bindings are passed, Hector will throw an error. If only a single binding is passed, Hector will merely perform a projection onto the target variables. If only two bindings are passed we have nothing to gain from the worst-case optimal strategy, because for each of the source bindings there will only be the one remaining binding left to propose anything. Similarly, we do not benefit from delta queries here, because a two-way join would not create any redundant intermediate arrangements.

For the general-arity case, Hector first employs the delta query technique. This means that a separate dataflow will be constructed for each binding that may experience change. By default, all attributes are assumed to experience change. Within each delta query, we refer refer to the generating binding as the *source binding*. Review Figure 2 for an illustration. The outputs of all delta queries created for a single Hector stage are concatenated together. After the final stage, an additional projection onto the variables requested by the user is performed.

Each delta query is a sequence of the worst-case optimal dataflow join primitive illustrated in Figure 5. Each stage extends changes on the source binding

to the next target variable. Picking an appropriate variable order is discussed in subsection 13.3. Here Delta-BiGJoin differs somewhat from non-incremental worst-case optimal join algorithms, in that it starts off with prefixes binding two variables (the elements of the source binding), whereas non-incremental algorithms start from the empty prefix. Dataflow computations must always start with some source of input. In our case, the finest grained source of input available are attributes, which already bind two symbols.

In order to get away with this in the general case, we must make sure to handle conflicts on the variables of each source binding. A conflict occurs whenever a non-source binding or constraint has opinions on either of the source variables but would never get a chance to enforce these opinions. A common example of this is a constant binding on either of the source variables. The constant binding will effectively be ignored. Reverse bindings such as `[?a :knows ?b] [?b :knows ?a]` cause a similar problem, because both of their variables are already bound by the other. We detect conflicts by looking for bindings for which all of their variables are already bound by the prefix. In our current implementation, Hector detects all conflicts, but only handles those with constant bindings. This is done by filtering the source binding on the constant value.

For the following we will again assume that a suitable variable order is at hand. We look at the variable order, and the variables bound by the current prefix and determine from that the next variable x, to which prefixes should be extended. Ignoring the source binding, we then filter all other bindings down to only those that bind ("talk about") x. From that set of relevant bindings we derive a set of *prefix extenders*. Each binding type corresponds to a prefix extender implementation. Our extender implementations are detailed in subsection 13.2. Extenders participate in the three-step process illustrated in Figure 5:

1. **Count** In the count step we determine for each source prefix the extender that will propose the fewest extensions for it. This is done by passing a stream of (prefix, count, extender)-triples through each extender's `count` operator. Initially all counts are set to infinity. The count step results in a stream of *nominations* for each extender (prefixes for which the respective extender has declared the fewest extensions).

2. **Propose** Nomination streams are passed through their respective extender's `propose` operator. Propose implementations materialize the extensions they counted in the previous step and output a stream of `(prefix extension)` pairs.

3. **Validate** Each extender's proposals are validated by all other extenders. This is done by passing the stream of proposed pairs through each extender's `validate` operator. Intuitively, validate implementations compute the intersection of the set of proposals with extensions that they themselves would have proposed, had they been nominated to extend.

Assuming that the count, propose, and validate implementations for prefix extenders satisfy certain conditions, each extensions will only do work on the

order of the worst-case result set bound. With the basic operation of the Hector operator covered, we now describe our prefix extender implementations and how they satisfy the requirements for worst-case optimality.

13.2 Prefix Extenders

From the discussion in Veldhuizen (2012) we learn that the Leapfrog Triejoin worst-case optimal join algorithm can be utilized to implement many common features of relational query engines: conjunctions (joins), disjunctions (unions), negations (antijoins), and filtering by various predicates. This is achieved by additional implementations of the core trie iterator interface. Similar extensions exist for Delta-BiGJoin.

Prefix extenders must provide `count`, `propose`, and `validate` operators. While conceptually simple, these implementations must not break the worst-case optimality, meaning that they can only do work proportional to the lowest number of extensions proposed by any of them (for a given extension stage).

The count operator must therefore avoid materializing and counting all proposals, as doing so for every extender would immediately violate the worst-case optimality. Implementations like Leapfrog TrieJoin and our own will accept count implementations that take logarithmic time (e.g. looking up prefixes in a suitable index). As long as only the winning extender is asked to propose, the propose implementation must only make sure to propose tuples with constant delay. A naive validation implementation might materialize all of its proposals and compute the set intersection. This of course violates worst-case optimality. However we may again take logarithmic time to lookup proposed extensions in an index.

CollectionExtender McSherry (c) gives an implementation of a prefix extender for Differential collections of key-value pairs, which we have largely adapted to implement prefix extension for 3DF attributes. The resulting implementation maintains attributes across three separate indices, corresponding to the count, propose, and validate operations. The count index maintains the number of distinct values for each key, the propose index maps keys to values, and the validate index maintains key-value pairs. Each index is implemented as a Differential arrangement, which support lookups in logarithmic time.

ConstantExtender Prefix extension for constant bindings is not backed by data other than the constant value. The count implementation simply issues a count of one for each prefix. Propose proposes for each prefix the constant value. Finally validate filters the stream of proposed extensions to only include those which match the constant value. These operations trivially satisfy the complexity bounds.

BinaryPredicateExtender Prefix extension for binary predicate bindings is again not backed by data. Predicate extenders can never be asked to propose, because they would have to propose infinitely many extensions. The count implementation is therefore a noop, whereas the propose implementation causes a runtime error, should the predicate extender receive a nomination. The validate implementation will apply the desired predicate to each proposed extension and

to the prefix offset binding the second argument. The supported binary predicates can be evaluated in constant time.

AntijoinExtender Antijoin extenders implement negation as set difference and thus wrap any of the other extenders. Due to the stratification requirement, antijoin extenders must not be asked to propose, and thus implement count as a noop and propose as a runtime error. During validation, antijoin extenders will subtract all extensions that the wrapped extender has validated from the result stream. Therefore, as long as the wrapped extender satisfies worst-case optimality, the antijoin extender will too.

These new indexing requirements had to be integrated into 3DF. Throughout this work we rely on base relations that are maintained in an indexed form and that can be shared across all dataflows. As described in this section, worst-case optimal prefix extenders require multiple different indices, in order to meet their complexity bounds. Additionally, whenever attributes are placed at a stage in a delta query, either of their symbols might be bound by the prefixes at that point. Therefore, reversed versions of all extender indices must be maintained as well.

We introduced a `CollectionIndex` structure which holds all three arrangements for a given direction and a given attribute. 3DF workers maintain separate mappings from attribute names to their forward and reverse collection indices. This simplifies the types and ownership involved. Collection indices implement a subset of Differential's arrangement API: an `import` method for importing all internal traces into a top-level scope, as well as an `enter_at` method to bring all imported internal arrangements into a nested scope with an additional timestamp coordinate.

Two additional implementation details become relvant in practice: picking a suitable variable order and preventing redundant outputs during concurrent execution of delta queries. We will discuss these in the following two sections.

13.3 Suitable Variable Orderings

We have deferred the problem of choosing a suitable variable ordering. While different orderings do not affect the theoretical guarantees, they can make a significant difference in practice. A suitable ordering is one that includes all variables bound by any of the participating bindings and which ensures that for each variable v there exists at least one binding that binds v. Additionally, we require that at least one of the (non-unary) bindings on v has its other side already bound by the prefixes at that point. This is analogous to avoid two-way joins on disjoint relations. While this can be limiting in edge cases, it simplifies the implementation and helps avoid pathological orderings in practice. Given the following clauses:

```
[?a :knows ?b]
[?b :knows ?c]
[?c :knows ?d]
[?a :knows ?d]
```

`[?a ?b ?c ?d ?e]` would be a suitable order, whereas `[?a ?c ?b ?d]` wouldn't, because none of the bindings can extend the prefix `[?a]` to `?c`.

However starting with the prefix `[?a ?b]`, it would be equally valid to extend first to `?c` and then to `?d` or vice versa. Deciding between the two is the problem of finding an *optimal* variable ordering. This question is significantly more involved and beyond the scope of this work. It is treated extensively in Abo Khamis et al. (2016).

13.4 Lexicographic Timestamps

All delta pipelines are executed concurrently at the dataflow level. This can lead to inconsistencies as we might derive the same output change on multiple pipelines, when sources change concurrently. Assume for example we have a graph, initially containing only a single edge $(100, 300)$, and two delta pipelines:

```
(I) d(a,b) -> (b,c) -> (a,c)
(II) d(b,c) -> (a,b) -> (a,c)
```

At some logical time t we then input two new edges $((100, 200), t, +1)$ and $((200, 300), t, +1)$. Crucially, both inputs are then logically available in all relations at time t. Along pipeline (I), the input edge $(100, 200)$ will therefore match the indexed edge $(200, 300)$ and thus cause the derivation of a single output change $((100, 200, 300), t, +1)$. Correspondingly, along pipeline (II) the input edge $(200, 300)$ will match the indexed edge $(100, 200)$, leading to a now redundant output change $((100, 200, 300), t, +1)$.

To prevent this, we must impose an arbitrary logical order on the computation. In particular, we must ensure that input changes are not logically available at all relations simultaneously (even though physically, they might already be indexed). For this we fix an arbitrary sequential order, e.g. $(a, b), (b, c), (a, c)$.

Delta pipelines are therefore created within a new, nested scope using a modified, lexicographic timestamp type. Timestamps t along each pipeline are extended to $(t, a \in 0, 1)$, where a is a boolean indicating the relative position of the source binding and the extending relation. These extended timestamps are ordered lexicographically, ensuring in particular that $(t, 0) < (t, 1)$. Attribute arrangements are imported and wrapped with their corresponding timestamps. We cache imported arrangements to prevent redundant imports.

Revisiting the scenario from above, the input edges would now be logically available in relation (b, c) on pipeline (II), but not on pipeline (I), because (b, c) appears after (a, b) in the relation order. Accordingly, only pipeline (II) would now derive the correct output change. This concludes our discussion of all relevant details of the Hector operator.

14 Other Findings and Future Work

A field as vast as query optimization and the novel computational model offered by Differential Dataflow makes it impossible to cover all relevant aspects and questions to a satisfactory degree. In this chapter we will touch upon insights and questions for future research that were obtained over the course of this work, without necessarily fitting its primary theme.

14.1 Automated Policies

We have pointed out significant challenges to continuously evaluated, complex relational query processing in an interactive, multi-user environment. While we described, implemented, and evaluated effective remedies, we have stopped short of providing a fully automated framework for making planning decisions based on these insights. Providing a fully automated planner is of course essential to any real-world system and will thus be of great interest to us in future work.

14.2 Hybrid Approaches

Throughout this work we have discussed the severe downsides of cost-based optimization and have accordingly made efforts to build on entirely different primitives. Nevertheless, exploring the interplay between worst-case optimal algorithms and traditional join processing within a hybrid cost-model warrants attention. A recent paper by Mhedhbi and Salihoglu (2019) introduces such a hybrid planner architecture, with promising results.

14.3 Adaptive Dataflows

Early on in section 6.3 we described the eddy operator from Avnur and Hellerstein (2000). We believe that the primary use case of eddies, adaptive join orderings, is better served by data-parallel worst-case optimal join algorithms as implemented in this work, because they do not require additional synchronization logic and provide provably optimal bounds.

However, adaptive routing between dataflows could be a complementary technique to avoid the overheads of worst-case optimal join processing (both in terms of dataflow size and best-case latencies). In combination with flow control techniques such as those developed in Lattuada et al. (2016), this could allow us to safely re-introduce heuristic elements into the query planner.

We imagine an adaptive operator that can route between a worst-case optimal and a traditional join-at-a-time dataflow, feeding new inputs only as either route completes its previous inputs. After running both plans in parallel for some time, the worse performing dataflow could be slowly starved and ultimately shut down.

For somewhat predictable data sources, doing so could significantly reduce the overall dataflow size and complexity, thus supporting more concurrent users.

On data sources that undergo significant changes over time, doing so would not protect us from sudden catastrophic skews any longer.

14.4 Datalog Evaluation Strategies

At its heart (and mostly thanks to Differential's incremental computation model), 3DF achieves competitive performance using only a semi-naive, bottom-up evaluation strategy. Over the past decades, many more sophisticated Datalog evaluation strategies have been developed (e.g. Tekle and Liu (2011)).

Many of these strategies hinge on finding clause orders or introducing additional clauses, s.t. the resulting sequence of clauses is semantically equivalent to the input clauses, but materialize fewer results.

More work is needed to understand how strategies such as the magic-set transform or subsumptive tabling translate into the incremental dataflow model, and to what extent they overlap with worst-case optimal join evaluation.

14.5 Set vs Multiset Semantics

Over the course of this work we have changed 3DF to expose Differential's multiset semantics, rather than impose set semantics on top. The reasons for this are motivated equally by performance cosiderations as well as user experience concerns. Here we must for once disagree with Aref et al. (2015), who state that:

> In our experience, [multiset semantics] also makes queries harder for the end users to write and to reason about.

Queries involving grouped aggregations commonly lead to redundant tuples within each group. Consider for example a simple query summing the total salary cost per company

```
[:find ?company (sum ?salary)
 :where
 [?company :company/employee ?employee]
 [?employee :employee/salary ?salary]]
```

The implicit grouping and projection expressed by the `:find` clause can cause a lot of redundant values, as many employees will have the same salary. Under set semantics, the resulting sums will not match user expectations. Epxressed purely in Differential Dataflow, the results would be "correct".

Before this work, 3DF had adopted the `:with` construct from the Datomic query language. Re-writing the same query as

```
[:find ?company (sum ?salary)
 :with ?employee
 :where
 [?company :company/employee ?employee]
 [?employee :employee/salary ?salary]]
```

would ensure that two employee salaries equal in value would still be considered as distinct tuples. Both, imposing set semantics on top of Differential (via the `distinct` operator), as well as implementing the `:with` construct meant introducing additional operators into the dataflow. The `distinct` operator in particular is stateful and must keep all of its inputs in arranged form.

Using multiset semantics also allows us to aggregate multiplicites directly. Differential can operate on multiplicites efficiently and in-place, whereas operations on data itself must remain immutable (and thus allocate new tuples). In the above example, 3DF will treat employee salaries as multiplicites of company eids $(((company, employee, 12345), t, 1)$ becomes $((company), t, 12345))$. Using Differential's built-in `count` operator, these multiplicites will be consolidated and accumulated in-place, according to any binary, associative operation under which the multiplicites form a monoid. Exploiting algebraic structure within the multiplicities to express more interesting aggregations is a topic of ongoing research (Abo Khamis et al. (2016)) and will be of great interest to us in the future.

On the input side however, users will expect set semantics, because that matches the mental model of a domain as individual facts. Transacting a triple such as `[x :knows y]` twice is usually expected to simply assert that fact *no matter what was true before*, rather than to indicate an increased weight of that specific edge in a weighted hypergraph.

The Hector operator in fact assumes its inputs to be distinct. For tuples with multiplicities larger than one, output multiplicites will not be correct any longer, because they might be multiplied more than once. This is because prefix and proposal multiplicites are multiplied *at each extension step*.

Unless otherwise specified by the user, 3DF will therefore apply a `distinct` to attribute input streams.

One underexplored aspect of this decision is how it affects the size of the available plan space. Quoting again Aref et al. (2015):

> [...] traditional SQL-based relational database systems adhere to a bag (multiset) semantics, [...]. This design choice drastically reduces the opportunity for query optimization, since it often happens that queries that are logically equivalent under set semantics are not under the bag-semantics.

We recognize this as a potential trade-off, but haven't yet explored its effects on our specific setting.

14.6 Richer Attribute Semantics

Throughout the discussions in chapter 9 in particular, we have made no stricter assumptions on attributes than that they be distinct (as touched on in the previous section). We suspect that allowing users to express more specific attribute semantics will open up entirely new optimization avenues.

We have often encountered attributes that should hold at most a single value for each distinct entity id. For example, an attribute such as `:device/speed`

might often be intended to hold the most recent measurement for each device. Assuming an existing fact `[x :device/speed 120]`, users expect a subsequent input `[x :device/speed 200]` to mean both the retraction of the previous value, as well as the assertion of the current one.

To accomodate this, we have extended 3DF with the concept of *input semantics*, which can be configured for each attribute individually. A custom stateful operator enforcing retraction of previous values is applied to attributes marked `CardinalityOne`.

Whenever a CardinalityOne attribute participates in a worst-case optimal prefix extension of its key symbol to its value symbol, we know that it will only ever propose zero or one. For star-joins on exclusively CardinalityOne attributes we can thus guarantee domain overlap and a uniform distribution of values. While attributes might still be correlated in adversarial ways, it might still be safe to use simpler, non worst-case optimal join implementations (delta queries without an estimation phase, or join-at-a-time pipelines).

Identifying, evaluating, and expressing such special cases within an extensible framework would be an interesting topic of further research.

15 Conclusions

We have investigated the problem of optimizing the continuous maintenance of relational queries for dynamic, multi-user environments. Besides the basic need for consistent, correct results, we identified robustness, support for many concurrent users with general-purpose use cases, and low-latency execution at high-throughputs as the driving goals behind this work. We thus were looking for techniques that could bring the domain of analytical, and graph databases closer to that of high-performance stream processing.

In closing, we want to summarize our findings and how they lead to a system that satisfies these properties.

1. The cyclic dataflow model in combination with progress tracking of partially-ordered logical times, as implemented by the Timely Dataflow framework, enables consistent, data-parallel computations that exploit fine-grained opportunities for concurrent execution. Support for local iteration of regions of the dataflow dramatically increase the class of algorithms that can be expressed within this model.

2. Distributed incremental computation with iteration and support for retractions, as implemented by the Differential Dataflow framework, makes it possible to efficiently maintain complex relational views over data streams.

3. Compact, shared representations of historical inputs, as implemented by Differential's arrangements, allow the dynamic creation of new dataflows with low-latency.

4. Declarative, relational languages such as Datalog, in combination with an attribute-oriented data model, are expressive enough to support a wide range of use cases, and are easily extended to new domains, without fundamentally affecting the underlying runtime.

5. Deferred synthesis of query plans is essential in dynamic environments, where queries might be defined across multiple declarative interactions, and where partially constrained plans significantly impact the overall system performance.

6. Exploiting the linearity of the relational join operator via delta queries makes it possible to implement joins of arbitrary arity with a constant memory footprint and without significantly impacting latencies. Delta queries can be combined to form join operators that update incrementally in response to arbitrary input changes.

7. Worst-case optimal dataflow join algorithms are compatible with delta queries and enable robust, predictable performance irrespective of clause order. The performance of the resulting operator preserves near real-time performance characteristics, is competitive with traditional join-at-a-time

approaches on best- and average-case queries, and offers asymptotic improvements for a class of relevant, cyclic queries. Crucially, these properties are preserved on long-running dataflows, even as the underlying data distributions change, without the need for synchronized re-optimization.

8. Extensions beyond purely conjunctive queries can be implemented within the same worst-case optimal join framework. We gave implementations of binary predicates, stratified negation, and constant bindings. Combined with Differential's support for fixed-point iteration, these cover the needs of many functional-relational query languages. Similarly, we outlined an extension to unary relations, that allows remediating categorical data as a common source of skew.

9. Modeling client parameters via additional joins on parameter relations allows both the dynamic re-configuration of existing dataflows, as well as the multiplexing of multiple client's interests onto a single dataflow. The resources of that dataflow can then be shared without limiting optimization opportunities. More generally applicable re-use is still possible via arrangements and scales significantly better than creating individual dataflows for every client.

We identified areas for future work, in particular on automated policies and hybrid planners, that are required to tie these primitives into fully-formed systems. Nevertheless we believe that our findings and their implementation within 3DF provide strong foundations on which a new class of systems, combining the analytical power of databases with the performance characteristics of stream processors, can build.

References

Abo Khamis, Mahmoud, Hung Q Ngo, and Atri Rudra (2016), "Faq: questions asked frequently." In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 13–28, ACM.

Agrawal, Jagrati, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman (2008), "Efficient pattern matching over event streams."

Ammar, Khaled, Frank McSherry, Semih Salihoglu, and Manas Joglekar (2018), "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows." *Proceedings of the VLDB Endowment*, 11, 691–704.

Aref, Molham, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn (2015), "Design and implementation of the logicblox system." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1371–1382, ACM.

Avnur, Ron and Joseph M Hellerstein (2000), "Eddies: Continuously adaptive query processing." In *ACM sigmod record*, volume 29, 261–272, ACM.

Blakeley, Jose A, Per-Ake Larson, and Frank Wm Tompa (1986), "Efficiently updating materialized views." In *ACM SIGMOD Record*, volume 15, 61–71, ACM.

Chen, Jianjun, David J DeWitt, Feng Tian, and Yuan Wang (2000), "Niagaracq: A scalable continuous query system for internet databases." In *ACM SIGMOD Record*, volume 29, 379–390, ACM.

Ciucanu, Radu and Dan Olteanu (2015), "Worst-case optimal join at a time." Technical report, Technical report, Oxford.

Condie, Tyson, David Chu, Joseph M Hellerstein, and Petros Maniatis (2008), "Evita raced: metacompilation for declarative networks." *Proceedings of the VLDB Endowment*, 1, 1153–1165.

Erling, Orri, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz (2015), "The ldbc social network benchmark: Interactive workload." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 619–630, ACM.

Forgy, Charles L (1989), "Rete: A fast algorithm for the many pattern/many object pattern match problem." In *Readings in Artificial Intelligence and Databases*, 547–559, Elsevier.

Gjengset, Jon, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Morris (2018), "Noria: dynamic, partially-stateful data-flow for high-performance web applications."

In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 213–231.

Goebel, Nikolas (2018), "Declarative dataflow." URL `https://github.com/comnik/declarative-dataflow`.

Graefe, Goetz (1995), "The cascades framework for query optimization." *IEEE Data Eng. Bull.*, 18, 19–29.

Graefe, Goetz and David J DeWitt (1987), *The EXODUS optimizer generator*, volume 16. ACM.

Graefe, Goetz and William J McKenna (1993), "The volcano optimizer generator: Extensibility and efficient search." In *ICDE*, volume 93, 209–218.

Hickey, Rich (2013), "Datomic." URL `https://www.datomic.com/`.

Hirzel, Martin, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm (2014), "A catalog of stream processing optimizations." *ACM Computing Surveys (CSUR)*, 46, 46.

Ilyas, Ihab F, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga (2004), "Cords: automatic discovery of correlations and soft functional dependencies." In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 647–658, ACM.

Inc., Confluent (2019), "Confluent kafka documentation." URL `https://docs.confluent.io/`.

Lattuada, Andrea, Frank McSherry, and Zaheer Chothia (2016), "Faucet: a user-level, modular technique for flow control in dataflow engines." In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, 2, ACM.

Leis, Viktor, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann (2015), "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, 9, 204–215.

Lohman, Guy (2014), "Is query optimization a "solved" problem." In *Proc. Workshop on Database Query Optimization*, 13, Oregon Graduate Center Comp. Sci. Tech. Rep.

Markl, Volker, Peter J Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran (2007), "Consistent selectivity estimation via maximum entropy." *The VLDB journal*, 16, 55–76.

Markl, Volker, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic (2004), "Robust query processing through progressive optimization." In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 659–670, ACM.

McSherry, Frank (????a), "Dataflowjoin." URL `https://github.com/frankmcsherry/dataflow-join`.

McSherry, Frank (????b), "Differential dataflow." URL `https://github.com/TimelyDataflow/differential-dataflow`.

McSherry, Frank (????c), "Dogsdogsdogs." URL `https://github.com/TimelyDataflow/differential-dataflow/tree/master/dogsdogsdogs`.

McSherry, Frank (????d), "Timely dataflow." URL `https://github.com/TimelyDataflow/timely-dataflow`.

McSherry, Frank, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard (2013), "Differential dataflow." In *CIDR*.

Mhedhbi, Amine and Semih Salihoglu (2019), "Optimizing subgraph queries by combining binary and worst-case optimal joins." *arXiv preprint arXiv:1903.02076*.

Murray, Derek G, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi (2013), "Naiad: a timely dataflow system." In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 439–455, ACM.

Ngo, Hung Q, Ely Porat, Christopher Ré, and Atri Rudra (2012), "Worst-case optimal join algorithms." In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, 37–48, ACM.

Ngo, Hung Q, Christopher Ré, and Atri Rudra (2013), "Skew strikes back: New developments in the theory of join algorithms." *arXiv preprint arXiv:1310.3314*.

Selinger, P Griffiths, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price (1979), "Access path selection in a relational database management system." In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, 23–34, ACM.

Tekle, K Tuncay and Yanhong A Liu (2011), "More efficient datalog queries: subsumptive tabling beats magic sets." In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 661–672, ACM.

Terry, Douglas, David Goldberg, David Nichols, and Brian Oki (1992), *Continuous queries over append-only databases*, volume 21. ACM.

Veldhuizen, Todd L (2012), "Leapfrog triejoin: A simple, worst-case optimal join algorithm." *arXiv preprint arXiv:1210.0481*.