DBToaster: Agile Views in a Dynamic Data Management System

Oliver Kennedy*
EPFL
oliver.kennedy@epfl.ch

Yanif Ahmad
Johns Hopkins University
yanif@jhu.edu

Christoph Koch
EPFL
christoph.koch@epfl.ch

ABSTRACT

This paper calls for a new breed of lightweight systems – *dynamic data management systems (DDMS)*. In a nutshell, a DDMS manages large *dynamic data structures* with *agile, frequently fresh views*, and provides a facility for monitoring these views and triggering application-level events. We motivate DDMS with applications in large-scale data analytics, database monitoring, and high-frequency algorithmic trading. We compare DDMS to more traditional data management systems architectures. We present the DBToaster project, which is an ongoing effort to develop a prototype DDMS system. We describe its architecture design, techniques for high-frequency incremental view maintenance, storage, scaling up by parallelization, and the various key challenges to overcome to make DDMS a reality.

1. INTRODUCTION

Dynamic, continuously evolving sets of records are a staple of a wide variety of today's data management applications. Such applications range from large, social, content-driven Internet applications, to highly focused data processing verticals like data intensive science, telecommunications and intelligence applications. There is no one brush with which we can paint a picture of all dynamic data applications – they face a broad spectrum of update volumes, of update impact on the body of data present, and data freshness requirements. However, modern data management systems either treat updates and their impact on datasets and queries as an afterthought, by extending DBMS with triggers and heavyweight views [17, 33, 47, 48], or only handle small, recent sets of records in data stream processing [12, 25, 34, 37].

We propose dynamic data management systems (DDMS) that are capable of handling arbitrary, high-frequency or high-impact updates on a general dataset, and any standing queries (views) posed on the dataset by long-running applications and services. We base the design of DDMS on four criteria:

1. The stored dataset is large and changes frequently.
2. The maintenance of materialized views dominates ad-hoc querying.
3. Access to the data is primarily by monitoring the views and performing simple computations on top of them. Some updates cause events, observable in the views, that trigger subsequent computations, but it is rare that the data store is accessed asynchronously by humans or applications.
4. Updates happen primarily through an *update stream*. Computations triggered by view events typically do not cause updates: there is usually no feedback loop.

A DDMS is a lightweight system that provides large dynamic data structures to support declarative views of data. A DDMS is *agile*, keeping internally maintained views fresh in the face of dynamic data. Client applications primarily interact with a DDMS by registering callbacks for view changes, rather than by accessing views directly. A DDMS does not necessarily provide additional DBMS functionality such as persistency, transactions, or recoverability.

Compared to a classical DBMS, a DDMS differs in its reaction to updates. To minimize response times, updates must be performed immediately upon arrival, precluding bulk processing. This determines the programming model: compared to a DBMS, control flow is reversed, and the DDMS invokes application code, not vice versa.

An active DBMS [36] could simulate a DDMS through triggers, but is not optimized for such workloads, and even if support for state-of-the-art incremental view maintenance is present, performs very poorly. Thus, DDMS differ from active databases in their being optimized for different workloads. DDMS are optimized for event processing and monitoring tasks, while active database systems are optimized to support traditional DBMS functionality such as transactions, which are not necessarily present in DDMS.

Compared to a data stream processing system and particularly an event processing system (such as Cayuga [25], SASE+ [1]), DDMS have much larger states, which will usually have to be maintained in secondary storage, and require drastically different query processing techniques. In a stream processor, the queries reside in the system while the data streams by. In a DDMS on the other hand, the data state is maintained in the system while a stream of updates passes through (much more like an OLTP system).

Moreover, event and stream processors [12, 34, 37] support drastically different query languages which are designed to ensure that only very small state has to be maintained, using windows or constructs from formal language theory [44].

*Also Dept. of Computer Science, Cornell University.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA. .

DDMS views are often rather complex and expensive, including large non-windowed joins and aggregation. In general, we expect DDMS to support standard SQL. The query processing techniques most suitable for such workloads come from DBMS research – incremental view maintenance in particular – and update stream research [16] but do not scale to high-frequency view maintenance.

We present two classes of applications motivating the desiderata and design choices of a DDMS, and then discuss one application scenario in detail.

Large-scale data analytics, but not as a batch job. Large-scale data analytics in the cloud are mostly performed on massively parallel processing engines such as map/reduce. These systems are not databases, as some strata of the systems, scientific computing, and large-scale Web applications communities find important to emphasize. Nevertheless, our research community can play an important role in making such systems more useful and effective. Clearly, the last word on posing *queries* in such systems has not been said.

Map/reduce-like systems achieve scalability at the cost of response time and interactivity. However, there is an increasing number of important applications of large-scale analytics that call for more interactivity or better response times that allow for online use. Among large Web applications, examples include (social or other) network monitoring and statistics [31], search with interactive feedback [7], interactive recommendations, keeping personalized Web pages at social networking sites up to date [46], and so forth. Many of these applications are not yet mission-critical to Web applications companies, but are becoming increasingly necessary for establishing and maintaining a competitive advantage.

Large-scale data analytics is equally present in more classical business applications such as data warehousing and scientific applications. Take the case of data warehousing with real-time updates: as data warehouses become increasingly mission-critical to commercial and scientific enterprises, the importance of up-to-date analyses increases. Traditionally, OLAP systems are not optimized for frequent updating, and may be considerably out-of-date. A DDMS could dramatically improve the freshness of warehoused data.

A DDMS is well-suited for use in large-scale data analytics through its provision of large dynamic data structures as views, instead of forcing programmers to re-implement view computations manually on top of key-value stores, and its emphasis on simple lightweight systems as opposed to the use of monolithic DBMS. Continually fresh DDMS views at first seem at odds with the bulk update processing dogma of large scale analytics systems, but enable important applications that require interactivity or event processing.

Database monitoring. There is an ever-increasing set of use cases in which aggregate views over large databases need to be continuously maintained and monitored as the database evolves. These queries can be thought of as continuous queries on the stream of updates to a database. However, it is only moderately helpful to view this as a stream processing scenario since the queries depend on very large state (the database) rather than a small window of the update stream, and cannot be handled by data stream processing systems.

Examples include policy monitoring (e.g., to comply with regulatory requirements to monitor databases of financial institutions, say to detect money laundering schemes) [6], network security monitoring, aiming to detect sophisticated attacks that span extended time periods, and data-driven simulations such as Markov-Chain Monte Carlo (MCMC). MCMC is an extremely powerful paradigm for probabilistic databases [15]. Query processing with MCMC involves walking between database states, iteratively making local alterations to the database. Each database state encountered while doing this is considered a sample; a view is re-evaluated and statistics on its results are collected. The key technical database problem is thus to compute the view for as many samples as possible, as quickly as possible [45]. This is precisely the kind of workload that DDMS are designed for. Further examples of database monitoring can be found in certain forms of automated trading. In the following, we discuss one form of data-driven automated trading which may well prove to be a killer application for DDMS.

Algorithmic trading with order books. In recent years, algorithmic trading systems have come to account for a majority of volume traded at the major US and European financial markets (for instance, for 73% of all US equity trading volume in the first quarter of 2009 [20]). The success of automated trading systems depends critically on strategy processing speeds: trading systems that react faster to market events tend to make money at the cost of slower systems. Unsurprisingly, algorithmic trading has become a substantial source of business for the IT industry; for instance, it is the leading vertical among the customer bases for high-speed switch manufacturers (e.g., Arista [43]) and data stream processing.

A typical algorithmic trading system is run by mathematicians who develop trading strategies and by programmers and systems experts who implement these strategies to perform fast enough, using mainly low-level programming languages such as C. Developing trading strategies requires a feedback loop of simulation, back-testing with historical data, and strategy refinement based on the insights gained. This loop, and the considerable amount of low-level programming that it causes, is the root of a very costly *productivity bottleneck*: in fact, the number of programmers often exceeds the number of strategy designers by an order of magnitude.

Trading algorithms often perform a considerable amount of data crunching that could in principle be implemented as SQL views, but cannot be achieved by DBMS or data stream processing systems today: DBMS are not able to (1) *update their views at the required rates* (for popular stocks, hundreds of orders per second may be executed, even outside burst times) and stream engines are not able to (2) *maintain large enough data state* and support suitable query languages (non-windowed SQL aggregates) on this state. A data management system fulfilling these two requirements would yield a very substantial productivity increase that can be directly monetized – the holy grail of algorithmic trading.

To understand the need to maintain and query a large data state, note that many stock exchanges provide a detailed view of the market microstructure through complete bid and ask *limit order books*. The bid order book is a table of purchase offers with their prices and volumes, and correspondingly the ask order book indicates investors' selling orders. Exchanges execute trades by matching bids and asks by price and favoring earlier timestamps. Investors continually add, modify or withdraw limit orders, thus one may view order books as relational tables subject to high update

volumes. The availability of order book data has provided substantial opportunities for automatic algorithmic trading.

EXAMPLE 1.1. To illustrate this, we describe the Static Order Book Imbalance (SOBI) trading strategy. SOBI computes a volume-weighted average price (VWAP) over those orders whose volume makes up a fixed upper k-fraction of the total stock volume in both bid and ask order books. SOBI then compares the two VWAPs and, based on this, predicts a future price drift (for example a bid VWAP larger than an ask VWAP indicates demand exceeds supply, and prices may rise). For simplicity, we present the VWAP for the bids only:

```
select avg(b2.price * b2.volume) as bid_vwap
from   bids b2
where  k * (select sum(volume) from bids)
         > (select sum(volume) from bids b1
            where b1.price > b2.price);
```

Focusing on the k-fraction of the order book closest to the current price makes the SOBI strategy less prone to attacks known as *axes* (large tactical orders far from the current price that will thus not be executed but may confuse competing algorithms).

Given continuously maintained views for VWAP queries on bid and ask order books, an implementation of the SOBI strategy only takes a few lines of code that trigger a buy or sell order whenever the ratio between the two VWAPs exceeds a certain threshold. □

We return to the two desiderata for query engines for algorithmic trading pointed out above. For trading algorithms to be successful, (1) views such as VWAP need to be maintained and monitored by the algorithms at or close to the trading rate. However, (2) the views cannot be expressed through time-, row- or punctuation-based window semantics. This lends weight to the need for DDMS that support agile views on large, long-lived state.

Structure of the paper. There are many technical challenges in making a DDMS a reality. This paper initiates a study of DDMS and presents DBToaster, a prototype developed by the authors. The contributions of this paper are as follows: In Section 2, we further characterize the notion of a DDMS and discuss a state machine abstraction of DDMS that further clarifies the programming model and abstractions relevant to users of such systems. Section 3 presents the DBToaster view maintenance technique. It demonstrates how the state machine abstraction, which calls for the compilation and aggressive pre-computation of the state transition function (viewed differently, this is the set of update triggers that cause views to be refreshed), leads to new incremental query evaluation techniques. Section 4 discusses storage management in DBToaster. Section 5 discusses scaling up through parallelization. We conclude in Section 6.

2. DDMS ARCHITECTURE

We now examine the architecture of a DDMS, as illustrated in Figure 1. The core component of a DDMS is its runtime engine. Unlike a traditional database system where the same engine manages all database instances, each individual DDMS execution runtime is constructed around a

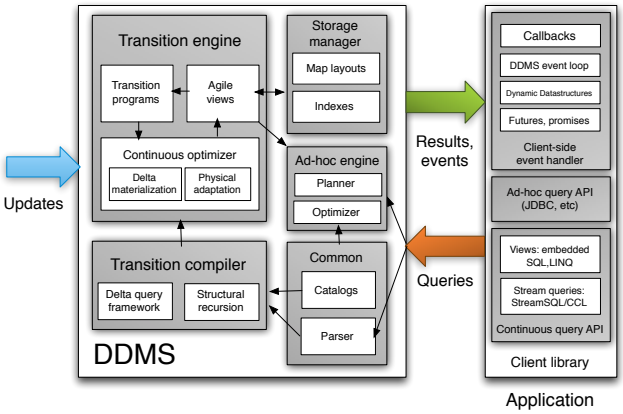

Figure 1: Dynamic Data Management System (DDMS) and Application Interface Architecture

specific set of queries provided by the client program (e.g., via SQL code embedded inline in the program), each defining an *agile view*.

2.1 Application Interfaces

The data that is processed by a DDMS arrives at the system in the form of an update stream of tuple insertions, deletions and modifications. The stream need not be ordered in any shape or form, and deletions are assumed to apply to tuples that have already been seen at some arbitrary prior point on the stream. Updates are fully processed on-the-fly, and their effects on agile views are realised in atomic fashion, prior to working on any subsequent update. Depending on the type of results requested by queries, any results arising from updates will be directly forwarded to application code as agile views are maintained.

DBToaster provides a wide variety of client interfaces to issue queries and obtain results from the DDMS, to reflect the diverse needs of applications built on top of our tool. Today's stream processors tend to be black-box systems that run completely decoupled from the application. Client libraries interact with stream processors through remote procedure call abstractions, issuing queries and new data through function calls, and either polling or being notified whenever results appear on a queue that is associated with a TCP socket connected to the stream processor.

In DBToaster, the set of agile views requested by clients, the *visible schema*, forms the primary read interface between client programs and the DDMS runtime. Clients can submit queries for which the DDMS materializes an agile view through three methods: (1) an embedded language, whose syntax and data model are natural fits to the host language in which the client application is written. Examples include embedded SQL, and collection comprehension oriented approaches such as LINQ, Links, and Ferry [11, 18, 29]. One interesting challenge with the embedded language approach is that of enabling asynchronous event-driven programming. Whereas language embeddings are natural for ad-hoc querying, we have yet to see these approaches for stream processing. This is the main mode of specifying queries that we focus on in this paper. (2) a continuous query client API, as done with existing stream client libraries, which sends a query string to the DDMS server for parsing, compilation, and agile view construction. The query string may be specified in a standard streaming language such as StreamSQL

or CCL [21]. The client may specify several ways to receive results, as seen below. (3) an ad-hoc query client API, which issues a one-time query to the DDMS, and returns the agile view as a datastructure to be used by the remainder of the client program. This API may be used in both synchronous and asynchronous modes, as indicated by the type of result requested. The query is specified in standard SQL.

Given these modes of issuing queries to the DDMS, our client interface supports four methods of receiving results: (1) callbacks, which can be specified as handlers as part of the continuous query API. Callbacks receive a stream of query results, and are the simplest form of result handlers that run to completion on query result events. (2) a DDMS event loop, which multiplexes result streams for multiple queries. Applications may register callbacks to be executed on any result observed on the event loop, allowing complex application behavior through dynamic registration, observation and processing of results on the event loop. (3) dynamic datastructures, which are read-only from the application perspective. The datastructure appears as a native collection type in the host language, facilitating natural access for the remainder of the program. Ad-hoc queries use this method for results by default. Continuous queries may also use this method in which case the datastructure acts as a proxy with accessors that pull in any updates from the DDMS when invoked. (4) promises and futures [26], which provide a push-based proxy datastructure for the result. A future is an object whose value is not initially known and is provided at a later time. A program using a query returning a future can use the future as a native datatype, in essence constructing a client-side dataflow to be executed whenever the future's value is bound. In our case, this occurs whenever query results arrive from the DDMS. Language embedded stream processing can be supported by futures, or program transformations to construct client side dataflow, such as continuation passing style as found in the programming languages literature [40].

2.2 DDMS Internals

The internals of the runtime engine itself are best viewed through the lens of a state machine. Compared to similar abstractions for complex event processors [1, 25], the state is substantially larger. Conceptually, the state represents an entire relational database and transitions represent changes in the base relations: events in the update stream.

Compiling transitions. Each transition causes maintenance work for our agile views, and just as with incremental view maintenance, this work can be expressed as queries. Maintenance can be aided by dynamic data structures, that is, additional agile views making up the *auxiliary schema*. A DDMS is a long-running system, operating on a finite number of update streams. This combination of characteristics naturally suggests *compiling* and specializing the runtime for each transition and associated maintenance performed by a DDMS. The transition compiler generates lightweight transition programs that can be invoked by the runtime engine with minimal overhead on the arrival of events. We describe the compiler in further detail in Section 3.

Storage management and ad-hoc query processing. Given the instantiation of an auxiliary schema and agile views, a DDMS must intelligently manage memory utilization, and the memory-disk boundary as needed. The storage manager of a DDMS is responsible for the efficient representation of both the agile views and any index structures required on these views. Section 4 discusses the issue of indexing, as well as how views are laid out onto disk. Supporting ad-hoc query processing turns out to be relatively straightforward given that the core of a DDMS continuously maintains agile views. Ad-hoc queries can be rewritten to use agile views in a similar fashion to the materialized view usage problem in standard query optimization. A key challenge here is how to ensure consistency, such that ad- hoc queries do not use inconsistent agile views as updates stream in and the DDMS performs maintenance. On the other hand, we do not want ad-hoc queries to block the DDMS' maintenance process and incur result delivery latency for continuous queries. One option here is to maintain a list of undo actions for each ad-hoc query with respect to agile view maintenance. This design is motivated by the fact that continuous queries are the dominant mode of usage, and ad-hoc queries are expected to occur infrequently, thus we bias the concurrency control burden towards ad-hoc queries.

Runtime adaptivity. Significant improvements in just-in-time (JIT) compilation techniques means that transition programs need not be rigid throughout the system's lifetime. A DDMS includes a compiler and optimizer working in harmony, leveraging update stream statistics to guide the decisions to be made across the database schema, state and storage. For example, the compiler may choose to compute one or more views on the fly, rather than maintaining it in order to keep expected space usage within predefined bounds. The optimizer's decisions are made in terms of the space being used, the cost of applying transitions on updates, as well as information from a storage manager that aids in physical aspects of handling large states, including implementing a variety of layouts and indexes to facilitate processing.

3. REALIZING AGILE VIEWS

Agile views are database views that are maintained as incrementally as possible. Despite more than three decades of research into incremental view maintenance (IVM) techniques [17, 35, 47, 48], agile views have not been realised, and one of our key contributions in handling large dynamic datasets is to exploit further opportunities for incremental computation during maintenance. Conceptually, current IVM techniques use delta queries for maintenance. Our observation is that the delta query is itself a relational query that is amenable to incremental computation. We can materialize delta queries as auxiliary views, and recursively determine deltas of delta queries to maintain these auxiliary views. Furthermore repeated delta transformations successively simplify queries.

3.1 View Maintenance in DBToaster

We present our observation in more detail and with an example. Given a query q defining a view, IVM yields a pair $\langle m, Q' \rangle$, where m is the materialization of q, and Q' is a set of delta queries responsible for maintaining m (one for each relation used in q that may be updated). DBToaster makes the following insight regarding IVM: current IVM algorithms evaluate a delta query entirely from scratch on every update to any relation in q, using standard query processing techniques. DBToaster exploits that a delta query q' from set Q' can be incrementally computed using the same principles as for the view query q, rather than evaluated in full.

To convey the essence of the concept, IVM takes q, pro-

duces $\langle m, Q' \rangle$ and performs m `+=` $q'(u)$ at runtime, where u is an update to a relation R and q' is the delta query for updates to R in Q'. We call this one step of *delta (query) compilation*. This is the extent of query transformations applied by IVM for incremental processing of updates. DBToaster applies this concept *recursively*, transforming queries to *higher-level deltas*. DBToaster starts with q, produces $\langle m, Q' \rangle$ and then recurs, taking each q' to produce $\langle m', Q'' \rangle$ and repeating. Here, each m' is maintained as m' `+=` $q''(v)$, where v is also an update, (possibly) different from u above, and q'' is the delta query from Q'' for the relation being updated. We refer to q' and q'' as first- and second-level delta queries respectively. We again recur for each q'', materialize it as m'', maintain it using third-level queries Q''', and so forth.

While delta queries are relational queries, they have certain characteristics that facilitate recursive delta compilation. First, DBToaster delta queries are parameterized SQL queries (with the same notion of parameter as in, say, Embedded SQL), with parameter values taken from updates. Thus, in particular, higher-level deltas are just (parameterized) SQL queries, but are not *higher-order* in the sense of functional programming as some queries in complex-value query languages are [8].

To illustrate parameters, we apply one step of delta compilation on the following query q over a schema `R(a int, b int), S(b int, c int)`:

$q =$ `select sum(a*c) from R natural join S`

For an update u that is an insertion of tuple $\langle$`@a`, `@b`$\rangle$ into relation R, the delta for q is:

$$q_R = \Delta_u(q) = \texttt{select sum(@a*c) from values(@a,@b), S where S.b = @b}$$
$$= \texttt{@a*(select sum(c) from S where S.b=@b)}$$

The `values (...)` clause is PostgreSQL syntax for a singleton relation defined in the query. Transforming a query into its delta form for an update u on R introduces parameters in place of R's attributes. We also apply a rewrite exploiting distributivity of addition and multiplication to factor out parameter `@a` from the query.

The second property that is key to making recursive delta processing feasible is that, for a large class of queries, delta queries are structurally strictly simpler than the queries that the delta queries are taken off. This can be made precise as follows. Consider SQL queries that are sum-aggregates over positive relational algebra. Consider positive relational algebra queries as unions of select-project-join (SPJ) queries. The *degree* of an SPJ query is the number of relations joined together in it. The degree of a positive relational algebra query is the maximum of the degrees of its member SPJ queries and the degree of an aggregate query is the degree of its positive relational algebra component. The rationale for such a formalization – based on viewing queries as polynomials over relation variables – is discussed in detail in [23]. It is proven in that paper that the delta query of a query of degree k is of degree $\max(k-1, 0)$. A delta query of degree 0 only depends on the update but not on the database relations. So DBToaster guarantees that a k-th level delta query $q^{(k)}$ has lower degree than a *(k-1)*-th level query $q^{(k-1)}$. Recursive compilation terminates when all conjuncts have degree zero.

Consider the delta query q_R above, which is of degree 1 while q is of degree 2. Query q_R is simpler than q since it does not contain the relation R. We can further illustrate the point by looking at a recursive compilation step on q_R. The second compilation step materializes q_R as:

$m_R =$ `select sum(c) from S where S.b=@b`

omitting the parameter `@a` since it is independent of the above view definition query. DBToaster can incrementally maintain m_R with the following delta query on an update v that is an insertion of tuple $\langle$`@c`, `@d`$\rangle$ into relation S:

$q_{RS} = \Delta_v(q_R) =$ `select @c from values(@c,@d)`

The delta query q_{RS} above has degree zero since its conjuncts contain no relations, indeed the query only consists of parameters. Thus recursive delta compilation terminates after two rounds on query q. In this compilation overview, we have not discussed the maintenance code for views m, m_R, and m_{RS} to allow the reader to focus on the core recursive compilation and termination concepts. We now discuss the data structures used to represent auxiliary materialized views, and then provide an in-depth example of delta query compilation including all auxiliary views created and the code needed to maintain these views.

Agile Views. DBToaster materializes higher-level deltas as agile views for high-frequency update applications with continuous group-by aggregate query workloads. Agile views are represented as main memory (associative) map data structures with two sets of keys (that is a doubly-indexed map $m[\vec{x}][\vec{y}]$), where the keys can be explained in terms of the delta query defining the map.

As we have mentioned, delta queries are parameterized SQL queries. The first set of keys (the *input* keys) correspond to the parameters, and the second set (the *output* keys) to the select-list of the defining query. In the event that a parameter appears in an equality predicate with a regular attribute, we omit it from the input keys because we can unify the parameter. We briefly describe other interesting manipulations of parameterized queries in our framework in the following section, however a formal description of our framework is beyond the scope of this paper.

Example. Figure 2 shows the compilation of a query q:

```
select l.ordkey, o.sprior, sum(l.extprice)
from   Customer c, Orders o, Lineitem l
where  c.custkey = o.custkey and l.ordkey = o.ordkey
group by l.ordkey, o.sprior
```

inspired by TPC-H Query 3, with a simplified schema:

```
Customer(custkey,name,nationkey,acctbal)
Lineitem(ordkey,extprice)
Order(custkey,ordkey,sprior)
```

The first step of delta compilation on q produces a map m. The aggregate for each group $\langle ordkey, sprior \rangle$ can be accessed as $m[][ordkey, sprior]$. We can answer query q by iterating over all entries (groups) in map m, and yielding the associated aggregate value. The first step also computes a delta query q_c by applying standard delta transformations as defined in existing IVM literature [17, 35, 47, 48]. In summary, these approaches substitute a base relation in a query with the contents of an update, and rewrite the query. For example, on an insertion to the `Customer` relation, we can substitute this relation with an update tuple $\langle$`@ck,@nm,@nk,@bal`$\rangle$:

Input (parent query)		Update	Output: auxiliary map, delta query		
$q =$	`select l.ordkey, o.sprior, sum(l.extprice) from Customer c, Orders o, Lineitem l where c.custkey = o.custkey and l.ordkey = o.ordkey group by l.ordkey, o.sprior;`	`+Customer (ck,nm,nk,bal)`	$m[][ordkey, sprior]$	$q_c =$	`select l.ordkey, o.sprior, sum(l.extprice) from Orders o, Lineitem l where @ck = o.custkey and l.ordkey = o.ordkey group by l.ordkey, o.sprior;`
q_c:	Recursive call, see previous output	`+Lineitem (ok,ep)`	$m_c[][custkey, ordkey, sprior]$	$q_{cl} =$	`select @ok, o.sprior,@ep*sum(1) from Orders o where @ck = o.custkey and @ok = o.ordkey`
q_{cl}:	Recursive call, see previous output	`+Order (ck2,ok2,sp)`	$m_{cl}[][custkey, ordkey, sprior]$	$q_{clo} =$	`select @sp, count() where @ck = @ck2 and @ok = @ok2;`

Figure 2: Recursive query compilation in DBToaster. For query q, we produce a sequence of materializations and delta queries for maintenance: $\langle m, q' \rangle, \langle m', q'' \rangle, \langle m'', q''' \rangle$. This is a partial compilation trace, our algorithm considers all permutations of updates.

```
select l.ordkey, o.sprior, sum(l.extprice)
from   values (@ck,@nm,@nk,@bal)
         as c(custkey,name,nationkey,acctbal),
       Orders o, Lineitem l
where  c.custkey = o.custkey and l.ordkey = o.ordkey
group by l.ordkey, o.sprior
```

Above the substitution replaces the `Customer` relation with a singleton set consisting of an update tuple with its fields as parameters. We can simplify q_c as:

```
q_c =  select   l.ordkey, o.sprior,sum(l.extprice)
       from    Orders o, Lineitem l
       where    @ck = o.custkey
       and      l.ordkey = o.ordkey
       group by l.ordkey, o.sprior;
```

The query rewrite replaces instances of attributes with parameters through variable substitution, as well as more generally (albeit not seen in this example for simpler exposition of the core concept of recursive delta compilation), exploiting unification, and distributivity properties of joins and sum aggregates to factorize queries [23].

This completes one step of delta compilation. Our compilation algorithm also computes deltas to q for insertions to `Order` or `Lineitem` (i.e. q_o and q_l). We list the full transition program for all insertions at the end of the example (deletions are symmetric, and ommitted due to space limitations). IVM techniques evaluate q_c on every insertion to `Customer`. To illustrate the recursive nature of our technique, we walk through the recursive compilation of q_c to m_c, q_{cl} on an insertion to `Lineitem` (see the second row of Figure 2). At this second step, DBToaster materializes q_c with its parameter `@ck` and group-by fields as $m_c[][custkey, ordkey, sprior]$, and uses this map m_c to maintain the query view m:

```
on_insert_customer(ck,nm,nk,bal):
  m[][ordkey,sprior] += m_c[][ck,ordkey,sprior];
```

As it turns out, all maps instantiated from simple equijoin aggregate queries such as TPCH Query 3 have no input keys. Maps with input keys only occur as a result of inequality predicates and correlated subqueries, for example the VWAP query from Section 1.

Above, we have a trigger statement in a C-style language firing on insertions to the `Customer` relation, describing the maintainence of m by reading the entry $m_c[\texttt{ck}, ordkey, sprior]$ instead of evaluating $q_c(\texttt{ck}, Orders, Lineitem)$. Notice that the trigger arguments do not contain *ordkey* or *sprior*, so where are these variables defined? In DBToaster, this statement implicitly performs an iteration over the domain of the map being updated. That is, map m is updated by looping over all $\langle ordkey, sprior \rangle$ entries in its domain, invoking lookups on m_c for each entry and the trigger argument `ck`. Map read and write locations are often (and for a large class of queries, always) in one-to-one correspondence, allowing for an embarrassingly parallel implementation (see Section 5). For clarity, the verbose form of the statement is:

```
on_insert_customer(ck,nm,nk,bal):
  for each ordkey,sprior in m:
    m[][ordkey,sprior] += m_c[][ck,ordkey,sprior];
```

Throughout this document we use the implicit loop form. Furthermore, this statement is never implemented as a loop, but relies on a map data structure supporting partial key access, or *slicing*. This is trivially implemented with secondary indexes for each partial access present in any maintenance statement, in this case a secondary index yielding all $\langle ordkey, sprior \rangle$ pairs for a given `ck`. This form of maintenance statement is similar in structure to the concept of *marginalization* in probability distributions, essentially the map m is a marginalization of map m_c over the attribute `ck`, for each `ck` seen on the update stream.

Returning to the delta q_{cl} produced by the second step of compilation, we show its derivation and simplification below:

```
select l.ordkey, o.sprior,
       sum(l.extprice)          select @ok, o.sprior,
from   Orders o, values                 @ep*sum(1)
       (@ok,@ep) as         =>  from   Orders o
       l(ordkey,extprice)       where  @ck = o.custkey
where  @ck = o.custkey          and    @ok = o.ordkey
and    l.ordkey = o.ordkey
```

Notice that q_{cl} has a parameter `@ck` in addition to the substituted relation `Lineitem`. This parameter originates from the attribute `c.custkey` in q, highlighting that map parameters can be passed through multiple levels of compilation. The delta q_{cl} is used to maintain the map m_c on insertions to `Lineitem`, and is materialized in the third step of compilation as $m_{cl}[][custkey, ordkey, sprior]$. The resulting maintenance code for m_c is (corresponding to Line 8 of the full listing):

```
on_insert_lineitem(ok,ep) :
  m_c[][custkey, ok, sprior] +=
    ep * m_cl[][custkey, ok, sprior];
```

Above, we iterate over each $\langle custkey, sprior \rangle$ pair in the map m_c, for the given value of the trigger argument `ok`. Note we have another slice access of m_{cl}, of $\langle custkey, sprior \rangle$ pairs for a given `ok`. The third step of recursion on insertion to

`Order` is the terminal step, as can be seen on inspection of the delta query q_{clo}:

```
select o.sprior, count()
from values (@ck2,@ok2,@sp)
as o(custkey,ordkey,sprior) =>
where @ck = o.custkey
and   @ok = o.ordkey
```

```
select @sp, count()
where @ck = @ck2
and   @ok = @ok2
```

In the result of the simplification, the delta q_{clo} does not depend on the database since it contains no relations, only parameters. Thus the map m_{cl} can be maintained entirely in terms of trigger arguments and map keys alone. Note this delta contains parameter equalities. These predicates constrain iterations over map domains, for example the maintenance code for q_{clo} would be rewritten as:

```
on_insert_order(ck2,ok2,sp) :
  m_cl[][ck, ok, sp] +=
    if ck==ck2 && ok==ok2
    then 1 else 0;
```

```
=> m_cl[][ck2, ok2, sp]
     += 1;
```

where, rather than looping over map m_{cl}'s domain and testing the predicates, we only update the map entry corresponding to `ck2`, `ok2` from the trigger arguments.

We show the trigger functions generated by DBToaster below, for all possible insertion orderings, including for example deltas of q on insertions to `Order` and then `Lineitem`. We express the path taken as part of the map name as seen for m_c and m_{cl} in our walkthrough. Some paths produce maps based on equivalent queries; DBToaster detects these and reuses the same map. Due to limited space, we omit the case for deletions, noting that these are symmetric to insertions except that they decrement counts.

```
1.  on_insert_customer(ck,nm,nk,bal) :
2.    m[][ordkey, sprior] +=
3.                      m_c[][ck, ordkey, sprior];
4.    m_l[][ordkey, sprior] +=
5.                    m_cl[][ck, ordkey, sprior];
6.    m_o[][ck] += 1;
7.
8.  on_insert_lineitem(ok,ep) :
9.    m[][ok, sprior] += ep *  m_l[][ok, sprior];
10.   m_c[][custkey, ok, sprior] +=
11.             ep *  m_cl[][custkey, ok, sprior];
12.   m_co[][ok] += ep;
13.
14. on_insert_order(ck,ok,sp) :
15.   m[][ok, sp]        += m_co[][ok] * m_o[][ck];
16.   m_l[][ok, sp]      +=  m_o[][ck];
17.   m_c[][ck, ok, sp]  += m_co[][ok];
18.   m_cl[][ck, ok, sp] += 1;
```

We briefly comment on one powerful transformation that is worth emphasizing in the above program, as seen on line 15. Notice that the right-hand side of the statement consists of two maps – all other statements are dependent on a single map. This line is an example of *factorization* applied as part of simplification. This statement is derived from the query q given at the start of the example, when considering an insertion to the `Order` relation with tuple $\langle$`@ck,@ok,@sp`$\rangle$:

```
q_o =  select   @ok,@sp,sum(l.extprice)
       from     Customers c, Lineitem l
       where    c.custkey = @ck
       and      l.ordkey = @ok;
```

The group-by clause of q can be eliminated since all group-by attributes are parameters. Note that the two relations `Customer`, and `Lineitem` have no join predicate, thus the query uses a cross product. By applying distributivity of the sum aggregate, we can separate (factorize) the above query into two scalar subqueries:

```
select @ok,@sp,
  sum(l.extprice)
from
  Customers c,
  Lineitem l
where c.custkey = @ck
and l.ordkey = @ok;
```

```
=>
```

```
select @ok,@sp,
((select sum(l.extprice)
  from   Lineitem l
  where  l.ordkey = @ok)
*
 (select sum(1)
  from   Customers c
  where  c.custkey = @ck));
```

DBToaster materializes the scalar subqueries above as m_{co} and m_o in its program, and in particular note that prior to factorization we had a single delta query of degree 2, and after factorization we have two delta queries of 1. The latter is clearly simpler and more efficient to maintain. Factorization can be cast as the generalized distributive law (GDL) [3] applied to query processing. GDL facilitates fast algorithms for many applications including belief propagation and message passing algorithms, and Viterbi's algorithm. With this analogy, we hope to leverage other techniques from this field, for example approximation techniques.

Transition program properties. For many queries, compilation yields *simple* code that has no joins and no nested loops, only single-level loops that perform probing as in hash joins. Simple code is beneficial for analysis and optimizations in machine compilation and code generation.

Transition programs leverage more space to trade off time by materializing delta queries. These space requirements are dependent on the active domain sizes of attributes, and often attributes do not have many distinct values, for example there are roughly 2800 unique stock ids on NASDAQ and NYSE. Additionally pruning duplicate maps during compilation facilitates much reuse of maps given recursion through all permutations of updates. Finally, there are numerous opportunities to vary space-time requirements for transitions: we need not materialize all higher-level deltas. For example we could maintain q with $m^{(i)}$, a materialized i-th level delta and perform more work during the update to evaluate $q^{(i-1)}, \ldots, q^{(1)}$. We could further amortize space by exploiting commonality across multiple queries, merging maps to service multiple delta queries.

Insights. Queries are closed under taking deltas, that is, a delta query is of the same language as the parent query. This allows for processing delta queries using classical relational engines in IVM. However, the aggressive compilation scheme presented above allows to innovate in the design of main-memory query processors. In the above example, we have materialized all deltas, thus the transition program consists of simple arithmetics on parameters and map lookups.

Our concept of higher-level deltas draws natural analogies to mathematics. Our framework, and the compilation algorithm described here – but restricted to a smaller class of queries without nested aggregates – was described and proven correct in [23]. In this framework, queries are based on polynomials in an algebraic structure – a *polynomial ring* – of generalized relational databases. This quite directly yields the two main properties that make recursive compilation feasible – that the query language is closed under taking deltas and that taking the delta of a query reduces its degree, assuring termination of recursive compilation. Unfortunately, the second property is not preserved if one extends the framework of [23] by nested aggregates. We describe be-

low how DBToaster handles this generalized scenario.

Discussion. To summarize, in contrast to today's IVM, DBToaster uses materialization of higher-level deltas for continuous query evaluation that is *as incremental as possible*. DBToaster is capable of handling a wide range of queries, including, as discussed next, nested queries. This has not been addressed in the IVM literature, and lets our technique cover complex, composed queries, where being as incremental as possible is highly advantageous.

3.2 Compilation Enhancements

We briefly discuss further compilation issues and optimizations beyond the fairly simple query seen in Figure 2.

Nested queries. We can compile transitions for nested queries, which has not been feasible in existing IVM techniques. In particular nested scalar subqueries used in predicates are problematic because taking deltas of such predicates does not result in simpler expressions. Our algorithm would not terminate if we did not handle this: we explicitly find simpler terms and recur on them. VWAP in Section 1 exemplifies a nested query.

Nested subqueries contain correlated attributes (e.g. price in VWAP) defined in an outer scope. We consider correlated attributes as parameters, or, internally in our framework, as binding patterns as seen in data integration. Nested queries induce *binding propagation*, similar to sideways information passing in Datalog. That is, we support the results of one query being used (or *propagated*) as the parameters of a correlated subquery, indicating an evaluation ordering. We transform queries to use minimal propagation, which performs additional aggregation of maps, over dimensions of the map key that are not propagated. For example a map $m[x, y, z]$ would be aggregated (*marginalized*) to $m'[x, y]$ if x, y were the only correlated attributes.

Rethinking query compilation with programming languages and compiler techniques. Our current compilation process involves implementing transition programs in a variety of target languages, including OCaml and C++. We currently rely on OCaml and C++ compilers to generate machine code, and observe that there are a wide variety of optimization techniques used by the programming languages (PL) and compiler communities that could be applied to query compilation. Compiling queries to machine code is not a novel technique, and has been applied since the days of System R [9]. However there have been many advances in source code optimization since then, as evidenced by several research projects aimed in this direction [4, 24].

We believe the advantage of incorporating methods from the PL and compiler communities directly into our compiler framework is that it facilitates whole-query optimizations, programmatic representation and manipulation of physical aspects of query plans such as pipelining and materialization (memoization), and opportunities to consider the interaction of query processing and storage layouts via data structure representations. Specifically, we have developed a small functional programming (FP) language as our abstraction of a physical query plan, unlike the operator-centric low-level physical plans found in modern database engines (e.g. specific join implementations, scans, sorting operators that make up LOLEPOPs in IBM's Starburst and DB2 [28], and similar concepts in Oracle, MS SQL Server amongst others).

The primary features of our FP language are its use of nested collections (such as sets, bags and lists) during query processing, its ability to perform structural recursion (SR) [8] optimizations on nested collections, and its support for long-lived persistent collections. Structural recursion transformations enable the elimination of intermediates when manipulating collections, and when combined with primitive operations on functions, such as function composition, yields the ability to adapt the granularity of data processing. Consider a join-aggregate query $\sum_{a*f}((R \bowtie S) \bowtie T)$ with schema $R(a, b)$, $S(c, d)$, $T(e, f)$, where the natural joins are Cartesian products. While such a query would not occur as a delta query in DBToaster (it would be factorized as discussed above), it suffices to serve as a toy example. Our functional representation is:

```
aggregate(fun < <t,u,v,w,x,y>, z>. (t*y)+z, 0,
  flatten(
    map(fun <w,x,y,z>.
          map(fun <e,f>.<w,x,y,z,e,f>, T),
        flatten(
          map(fun <a,b>.
                map(fun <c,d>.<a,b,c,d>, S),
              R)))))
```

Above, `fun` is a lambda form, which defines a function possibly with multiple arguments as indicated by the tuple notation. Next, `map`, and `aggregate` are the standard functional programming primitives that apply a function to each member of a collection, and fold or reduce a collection respectively. For example, we can use `map` to add a constant to every element of a list as:

```
map(fun x. x+5, [10;20;30;40]) => [15;25;35;45]
```

Similarly we can use aggregate to compute the sum of all elements of a list, with initial value 0 as:

```
aggregate(fun <x,y>. x+y, 0, [10;20;30;40]) => 100
```

The `flatten` primitive applies to a collection of collections (i.e. a nested collection), yielding a single-level collection. For example:

```
flatten([[1;2]; [3]; [4;5;6]]) = [1;2;3;4;5;6]
```

Our functional representation first joins relations R and S, before passing the temporary relation created to be joined with T, again yielding an intermediate, that is finally aggregated. Notice the intermediate flatten operations to yield a first normal form.

A standard implementation of a join as a binary operation forces the materialization of the intermediate result $R \bowtie_\theta S$. There are numerous scenarios where such materialization is undesirable, and has led to the development of multiway join algorithms such as XJoin [41] and MJoin [42]. With a few simple transformation steps, we can rewrite the above program to avoid this intermediate materialization as:

```
aggregate(fun < <t,u,v,w,x,y>, z>. (t*y)+z, 0,
  flatten(flatten(
    map(fun <a,b>. map(fun <c,d>. map(fun <e,f>.
      <a,b,c,d,e,f>, T), S), R))))
```

This is a three-way nested loops join $\bowtie_3$ (R, S, T) with no intermediate materialization of a two-way join, that can also be pipelined into the aggregation. In general, we can apply well-established folding [5], defunctionalization [13] and deforestation [27] techniques from functional programming, which when combined with data structures such as indexes

and hash tables, can yield a rich space of evaluation strategies that vary in their pipelining, materialization, ordering, nesting structure and vectorization characteristics.

The last item, nesting structure and vectorization, refers to concepts from nested relational algebra [38], whereby query processing need not occur in terms of first normal form relations. Our programs can use non-first normal forms internally, and can apply compression and vectorized processing over the nested relation attributes, much in the vein of column-oriented processing. Furthermore, we can directly represent the aforementioned data structures, such as indexes and DBToaster maps, in our language, yielding a unified approach to representing both query processing and storage layout. We know of no existing framework capable of such a rich representation, and are excited by the potential to apply program transformations to jointly optimize query processing and storage.

4. MANAGING STORAGE IN DBTOASTER

We now examine two components of DBToaster's solution to storage in a DDMS: (1) The DBToaster compiler produces data structures designed specifically for the compiled DDMS' target query workload. (2) By analyzing the patterns with which data is accessed, DBToaster constructs a data layout strategy (for pages on a disk, servers in a cluster, etc.) that limits IO overhead.

Data structures. DBToaster uses multi-key (i.e., multi-dimensional) maps to represent materialized views. The generated code performs lookups of slices of the maps, using partial keys, fixing some dimensions and iterating over the others. Recall the right-hand side of line 2 in the code listing in Section 3. This is a partial lookup on map m_c with only `ck` defined by the trigger arguments. In addition to exact and partial lookups, DBToaster maps support range lookups by inequality predicates.

In their simplest form, out-of-core maps are implemented by a simple relational-style key-value store with secondary indices [30]. Inequality predicates, and aggregations including such predicates, are implemented efficiently using maps that store cumulative sums [19]. Maps can apply compression techniques to address frequently repeating data. DBToaster customizes the data structures backing each materialized view based on statement-level information on accesses, applying static compile-time techniques to construct specialized data structures.

With substantial specialization of data structures as part of compiling transitions, DBToaster is free to consider a range of runtime issues in data structure tuning and adaptation, including how to best perform fine-grained operations such as incremental and partial indexing [39]. The key challenge to be addressed is how to provide data structures with a low practical update cost (avoiding expensive index rebalancing and hash-table rebucketing) while gradually ensuring the lookup requirements of our data structures are retained over time, amortizing data structure construction with continuous query execution.

Partitioning and co-clustering by data flow analysis. Database partitioning and co-clustering decisions are traditionally made based on a combination of (a) workload statistics, (b) information on schema and integrity constraints (such as key-foreign key constraints, a popular basis for co-clustering decisions), and (c) a body of expert insights into how databases execute queries. Ideally, such decisions should be based on a combination of (a), (b), and a *data flow analysis* of the system's query execution code, instantiated with the query plan, or view maintenance code. In classical DBMS however, this is too difficult to be practical.

Fortunately, data flow analysis turns out to be feasible for compiled DDMS transition programs: in fact, it is rather easy. A transition program statement reads from several maps and writes to one, prescribing dependencies between those maps occurring on the right-hand side of the statement (reading), and the one on the left-hand side (writing). As pointed out in Section 3, transition program statements admit a perfectly data-parallel implementation: consequently, a statement never imposes a dependency between two items of the same map and any horizontal partitioning across the involved maps map keeps updates strictly local. Using these data flow dependencies, partitioning and co-clustering decisions can be made by solving a straightforward min-cut style optimization problem.

5. DBTOASTER IN THE CLOUD

Scaling up a DDMS requires not only storing progressively more data, but also a dramatic increase in computing resources. As alluded to in Section 4, DDMS and their corresponding transition programs are amenable to having their data distributed across a cluster: (1) The only data structures used by transition programs are maps, which are amenable to horizontal partitioning. (2) At the granularity of a single update, iterative computations are completely data-parallel.

Update Processing Consistency and Isolation. In DBToaster, transition functions are created under the assumption that they are operating on a *consistent* snapshot of the DDMS's state. The entire sequence of statements composing the trigger function must be executed atomically, to ensure that each statement operates on maps resulting from fully processing the update stream prior to the update that fired the trigger. Thus the effects of updates should be fully isolated from each other. Similar issues and requirements have been raised before in the single-site context of view maintenance with the "state bug" [10].

Our requirement of processing updates in such an order is conservative, indeed we could apply standard serializable order concurrency control here to simultaneously process updates that do not interact with each other. The underlying goal is to develop simple techniques that avoid heavyweight locking and synchronization of entries in massively horizontally partitioned maps. We start with a conservative goal to focus on lightweight protocols. Ensuring this atomicity property is the first of the two core challenges that we encountered while constructing a distributed DDMS runtime.

Distributed Execution. Each update in our distributed DDMS runtime design employs three classes of actor:

- *source nodes*: Nodes hosting maps read by the update's trigger function (maps appearing on the right-hand side of the function's statements).
- *computation nodes*: The nodes where statements are evaluated.
- *destination nodes*: Nodes hosting maps written to by the update's trigger function (maps appearing on the left-hand side of the function's statements).

Note that these actors are logical entities; it is not necessary (and in fact, typically detrimental) for the actors to be on separate physical nodes within the cluster. Introducing a distinction between the different tasks involved in update processing allows us to better understand the tradeoffs involved in the second core challenge: selecting an effective partitioning scheme that intelligently determines placements of logical entities in order to best utilize plentiful hardware to handle a large update stream and DDMS state.

5.1 Execution Models

We first address the issue of atomicity by providing two execution models: (1) A protocol that provides a serial execution environment for transition programs, and (2) An eventual consistency protocol that provides the illusion of serial execution.

Serial Execution. The most straightforward way of achieving atomicity is to ensure serial trigger function execution. However, requiring all nodes in the cluster to block on a barrier after every update is not a scalable approach. A similar effect can be achieved more efficiently by using fine-grained barriers, where each update is processed by first notifying all of the update's destination nodes of an impending write. Reads at the update's source nodes are blocked while writes from prior updates are pending.

Serial execution requires a global ordering of updates as they arrive at the DDMS. Techniques to achieve this include:

- Updates arrive only from a single producer (e.g., the cluster is maintaining a data warehouse that mirrors a single OLTP database).
- A central coordinator generates a global ordering (as in [32]).
- A distributed consensus protocol generates a global ordering (as in [22]).
- A deterministic scheme produces a global ordering. For example, each update producer generates timestamps locally and identical timestamps in a global view are settled with a deterministic tiebreaker like the producer's IP address.

We also need a mechanism to provide consistent delivery of updates from multiple producers. Before completing a read, source nodes must not only ensure that all prior pending writes have been completed, but also that all notifications for prior updates have been received. A simple solution is to channel all updates through a single server. This has the advantage of also providing a global ordering over all updates. However this solution creates a scalability bottleneck. Alternative solutions like broadcasting updates or periodic commits are possible, but introduce considerable synchronization overheads.

Speculative Execution with Deltas. As an alternative, we can favor the use of speculative and optimistic processing techniques in designing our distributed execution protocol. The key insight here is that our computation is based on incremental processing, thereby unlike standard usage of speculative execution, any work done speculatively need not be thrown away entirely, rather any work done can be revised through increments or deltas, to the final desired outcome.

In particular, a node can optimistically perform reads immediately (or at least, blocking only on pending write operations which the node is already aware of, and not the vague possibility of potential future write operations). Although avoiding blocking on potential future writes eliminates significant synchronization overheads, out-of-order updates can cause the atomicity and desired ordering properties of trigger execution to be lost. We favor this point in the design space since out-of-order events are expected to occur infrequently. Furthermore, such events are likely to interfere with only a handful of prior updates, for example a write on one map entry followed by an out-of-order read on a different entry in the same map do not cause a problem. Finally, because write operations are limited to additive deltas, there is a clear mechanism for composing out-of-order writes.

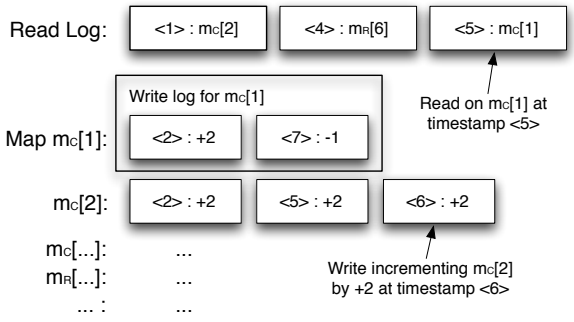

Figure 3: Supplemental data structures used to facilitate speculative execution in a distributed DDMS.

Out-of-Order Processing with Deltas as Revisions. Two types of out-of-order operations can occur in the speculative execution model: write-before-read, and read-before-write. We supplement maps with two additional data structures capturing timestamp information for operations, as illustrated in Figure 3: (1) Source nodes maintain a log of all read operations. (2) Destination nodes save all write operations independently; map entries are saved as logs rather than summed values. Each operation is tagged with and sorted by the effecting update's timestamp `<t>`.

In the case of an out-of-order read operation (i.e., one that arrives after a write operation that logically precedes it), the write log makes it possible to reconstruct the state of the map at an earlier point in time. For example, given the initial state in Figure 3, an update that requires a read on entry m_C`[2]` arrives with timestamp `<3>`. The value sent to the computation nodes is not the latest value of the entry (m_C`[2]` $= 6$ for all timestamps after `<6>`), but rather the sum of all values with lower timestamps (m_C`[2]` $= 2$ for timestamps `<3>`,`<4>`, and `<5>`).

In the case of an out-of-order write operation, the read log allows us to send a *revising update* to each computation node affected by the write. For example, given the initial state in Figure 3, an update that requires a write on entry m_R`[6]` arrives with timestamp `<3>`. The value will be written as normal (i.e., inserted into the write log for m_R`[6]`, in sorted timestamp order). Additionally, because the read log shows a read on the same entry with a later timestamp, a corrective update will be sent to the computation node(s) to which the entries were originally sent to.

Both data structures grow over time. To prevent unbounded memory usage, it is necessary to periodically truncate, or garbage collect the entries in each. This in turn, requires the runtime to periodically identify a cutoff point,

the "last" update for which there are no operations pending within the cluster. The read history is truncated at this point, and all writes before this point are coalesced into a single entry. Though this process is slow, it does not interfere with any node's normal operations, and can be performed infrequently, for example once every few seconds.

Hybrid Consistency. While the speculative execution model and its eventually consistent results are advantageous from a performance and scalability perspective, there may not be a point at which the state of all maps in the system corresponds to a consistent snapshot of our transition programs evaluated over any prefix of the update stream. That is, there is no guarantee that the system has actually converged to its eventually consistent state in the presence of a highly dynamic update stream. However, a side effect of the garbage collection process is that each garbage collection run, in effect generates a consistent snapshot of the system. As in other eventual consistency systems [14], this approach offers a hybrid consistency model, specifically the same infrastructure produces both low-latency eventually consistent results, as well as higher-latency consistent snapshots.

5.2 Partitioning Schemes

The second challenge associated with distributing a transition program across the cluster is the distribution of logical nodes (source, computation, and destination) across physical hardware in the cluster. In addition to more complex, min-cut based partitioning schemes for the data, DBToaster considers two simple partitioning heuristics for distributing computation: (1) data shipping: evaluate program statements where their results will be stored, at destination nodes, or (2) program shipping: evaluate program statements where their input maps are stored, at source nodes.

Destination-Computation. Given the one-to-one correspondence between computation nodes and destination nodes, the simplest partitioning scheme is to perform computations where the data will be stored – that is, the destination and computation nodes are co-located. As part of update evaluation, each source node transmits all relevant map entries to the destination node. Upon arrival, the destination node evaluates the statement and stores the result.

Source-Computation. Though simple, transmitting every relevant map entry with every update can be wasteful, especially if the input map entries don't change frequently. An alternative approach is to co-locate all of the source nodes and the computation node. When evaluating an update, the computation can be performed instantaneously, and the only overhead is transmitting the result(s) to the destination node(s). This is particularly effective in queries where update effects are small (e.g., queries consisting mostly of equijoins on key columns).

However, this approach introduces an additional complication. It is typically not possible to generate a partitioning of the data that ensures that for each statement in a trigger program, all the source nodes will be co-located. In order to achieve a partitioning, data must be replicated; each map is stored on multiple physical nodes. While replication is typically a desirable characteristic, storage-constrained infrastructures may need to use complex partitioning schemes.

6. DISCUSSION AND CONCLUSIONS

We have proposed Dynamic Data Management Systems, arguing for the need for a class of systems optimized for keeping SQL aggregate views highly available and fresh under high update rates. We have sketched some of the main research challenges in making DDMS a reality, and have outlined key design decisions and first results and insights that make us confident that the vision of DDMS can be realized.

We are currently developing a DDMS, DBToaster, in a collaboration between EPFL and Johns Hopkins, which was started while the authors worked at Cornell. So far, we have developed an initial version of a transition compiler which implements the recursive IVM technique sketched in Section 3. The feasibility of keeping views fresh through hundreds of updates per second using this approach in the context of algorithmic trading was recently demonstrated using an early DBToaster prototype [2]. An initial account of the foundations and theory of recursive IVM was given in [23]. Apart from substantially improving the compiler, we are now working on the actual DBToaster DDMS. We follow the strategy of developing two branches, one a main-memory, moderate state-size, single-core ultra-high view refresh rate system for applications such as algorithmic trading and the other a cloud-based, persistent, eventual-consistency system for very large scale interactive data analysis. We will merge these two branches once the individual technical problems have been solved. Further details and an update stream on the project can be found on our website at `http://www.dbtoaster.org`.

Acknowledgments

This project was supported by the US National Science Foundation under grant IIS-0911036. Any opinions, findings, conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of NSF. We thank the Cornell Database Group for their feedback throughout the development of the DBToaster project.

7. REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[2] Y. Ahmad and C. Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2), 2009.

[3] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2), 2000.

[4] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, 2010.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4), 1994.

[6] D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *CAV*, pages 1–18, 2010.

[7] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR&DB integration. In *CIDR*, 2007.

[8] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1), 1995.

[9] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for repetitive transactions and ad hoc queries in System R. *ACM Trans. Database Syst.*, 6, 1981.

[10] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, 2006.

[12] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2003.

[13] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP*, 2001.

[14] D.B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[15] D. Deutch, C. Koch, and T. Milo. On probabilistic fixpoint and Markov chain query languages. In *PODS*, 2010.

[16] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM TODS*, 35(1), 2010.

[17] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.

[18] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: database-supported program execution. In *SIGMOD*, 2009.

[19] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, 1997.

[20] R. Iati. The real story of trading software espionage. AdvancedTrading.com, July 2009.

[21] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik. Towards a streaming SQL standard. *PVLDB*, 1(2), 2008.

[22] F. P. Junqueira and B. C. Reed. The life and times of a zookeeper. In *PODC*, New York, NY, USA, 2009.

[23] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[24] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[25] L. Brenna et al. Cayuga: a high-performance event processing engine. In *SIGMOD*, 2007.

[26] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, 1988.

[27] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Functional Programming*, 1992.

[28] J. McPherson and H. Pirahesh. An overview of extensibility in starburst. *IEEE Data Eng. Bull.*, 10(2), 1987.

[29] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and xml in the .net framework. In *SIGMOD*, 2006.

[30] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX*, 1999.

[31] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.

[32] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[33] R. G. Bello et al. Materialized views in Oracle. In *VLDB*, 1998.

[34] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[35] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms & cost analysis. *ACM TODS*, 16(3), 1991.

[36] S. Ceri et al. Practical applications of triggers and constraints: Success and lingering issues. In *VLDB*, 2000.

[37] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[38] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2), 1986.

[39] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4), 1989.

[40] G. J. Sussman and G. L. S. Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4), 1998.

[41] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2), 2000.

[42] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

[43] A. von Bechtoldsheim. Scalable networking for cloud datacenters. Invited Talk, EPFL, September 2010.

[44] W. M. White, M. Riedewald, J. Gehrke, and A. J. Demers. What is "next" in event processing? In *PODS*, 2007.

[45] M. L. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *PVLDB*, 3(1), 2010.

[46] Y. Fu et al. AJAX-based report pages as incrementally rendered views. In *SIGMOD*, 2010.

[47] J. Zhou, P.-Å. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, 2007.

[48] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *ICDE*, 2007.