From a Stream of Relational Queries to Distributed Stream Processing

Qiong Zou
IBM Corporation
China Research Lab

Huayong Wang
IBM Corporation
China Research Lab

Robert Soulé
IBM Corporation
T. J. Watson Research Center
New York University

Martin Hirzel Henrique Andrade Buğra Gedik Kun-Lung Wu
IBM Corporation
T. J. Watson Research Center

ABSTRACT

Applications from several domains are now being written to process live data originating from hardware and software-based streaming sources. Many of these applications have been written relying solely on database and data warehouse technologies, despite their lack of need for transactional support and ACID properties. In several extreme high-load cases, this approach does not scale to the processing speeds that these applications demand. In this paper we demonstrate an application acceleration approach whereby a regular ODBC-based application is converted into a true streaming application with minimal disruption from a software engineering standpoint. We showcase our approach on three real-world applications. We experimentally demonstrate the substantial performance improvements that can be observed when contrasting the accelerated implementation with the original database-oriented implementation.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*

General Terms

Continuous processing, streaming processing, ODBC

1. INTRODUCTION

Data analytics are becoming a front-and-center issue in many areas of business and science. One of the driving forces behind this movement is the need for business analysts, decision makers, and scientists alike to have better tools to make more accurate management decisions or to extract, on a timely basis, better insights from data. These sources continuously generate new data and are an integral part of the increasingly complex information technology infrastructure that forms the underpinnings of modern businesses and scientific instruments. A second important trend is that, as a result of better instrumentation and business growth, more data is becoming available (both in volume and in data rates), so there is more to analyze. Furthermore, competition in the market-place requires better analysis as a critical strategic necessity, either to improve the overall efficiency of the business or to lower costs associated with waste and fraud.

These two trends have been around for some time as part of the natural cycle of re-invention that surrounds businesses and the basic information technologies that innovative businesses rely on. In reality, a lot of applications implemented by companies and research institutions can be construed as *data analytics* applications. The fundamental shift that is happening over the last few years, however, is the availability of more flexible tools for implementing this breed of applications [11, 35, 39] such that they can scale to the data volumes and rates that are now becoming common and can provide *timely* responses.

Most of these applications are implemented using a model where they continuously receive and process live data, and in response, among other processing steps, originate one or more queries against a database or data warehouse. Note, however, that a lot of the strengths of databases are actually not used in these cases, since a lot of this data is the representation of (periodically) pre-computed models, is very static, and, in many cases, read-only or with much looser synchronization and locking needs. When data rates and volumes are small, this approach works well and thus is widely employed in the industry. However, when volumes and rates increase, a natural slowdown occurs as database operations become a bottleneck, as we will experimentally demonstrate later.

This paper proposes a framework whereby applications built according to the model we outlined above can be semi-transparently converted from an application that generates a stream of queries into a distributed streaming application. Indeed, the system infrastructure we designed allows applications to be minimally disrupted from a code base standpoint. In essence, the points where frequent database interactions are made are converted into interactions with a streaming infrastructure that is automatically generated to manage the data at rest and process queries on it in a

Proceedings of the VLDB Endowment, Vol. 3, No. 2
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

distributed fashion.

As we will show later in this paper, employing our framework typically leads to substantial performance improvements over the original implementations. Moreover, our approach provides the means for a natural scale-out with the addition of more computational resources. In particular, our contributions are: (1) A mechanism for converting a database-based (or DB-based, for short) data analysis application into a streaming application, by semi-automatically transforming the embedded DB-type queries and the data at rest; (2) An extensive experimental evaluation including several case study applications as well as implementation alternatives varying from the original implementation, to an in-memory DB-based one, as well as the *transformed* application using our framework.

2. PROBLEM STATEMENT

An emerging class of applications comprises those that are continuous in nature in that external sources drive the computation by generating a stream of events. Over the last few years, stream computing middleware has been developed to address the particular needs of these types of applications [10]. Nevertheless, most of these applications are not purely streaming, as most of them must rely on information that has been accumulated over time and distilled into models that incorporate historic patterns in aggregated form. Indeed, a common architectural pattern is to use models computed from the accumulated data (or *data at rest*) to *score* and *enrich* the streaming incoming data (or *data in motion*). While these applications are reasonably common in several domains (see examples in Section 3), in most cases, they are implemented using relational database and data warehousing technologies. In other words, even the stable and read-only data used for scoring the incoming events is kept in relational databases, despite the lack of need for typical relational database transactional and logging guarantees. As we will show, the reliance on conventional database servers considerably hinders these applications' response time and scalability.

With these considerations in mind, we believe that a framework that can *semi-transparently* inspect an application's source code, identify the queries against large and infrequently updated datasets, and transform these queries into a distributed query network against a partitioned version of the original dataset can substantially improve the application's response time and throughput.

3. CASE STUDY APPLICATIONS

In this section we describe three applications, which have been deployed in customer environments. These applications are also undergoing further development where scalability requirements are being addressed to cope with load imposed by increased usage.

These applications have a common underlying architecture, including the streaming nature of their primary workload as well as their reliance on ODBC calls for querying large scale, mostly read-only datasets. In our discussion we will include the SQL formulation for the specific bottleneck query in each of these applications. As will be seen in Section 6, these queries are the focal point of the experimental study we used to evaluate our application transformation framework.

3.1 Trajectory Mapping

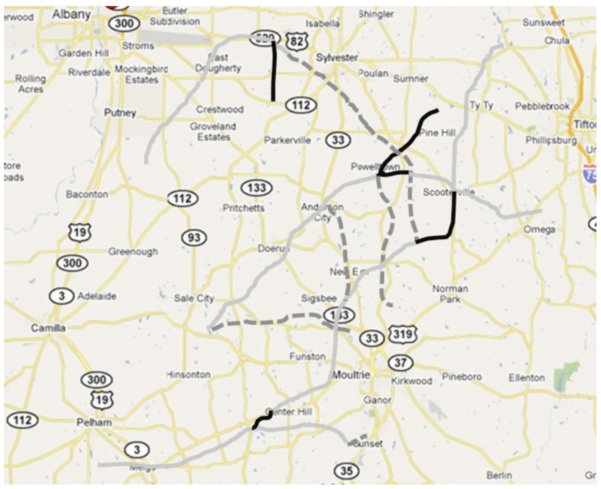


Figure 1: The Trajectory Mapping application results overlaid on Google Maps. Black solid lines indicate the most congested road segments.

Intelligent Transportation Systems (ITS) aim at continuously tracking vehicle movement to optimize live vehicle routing decisions as a means to decrease travel times and reduce fuel consumption. In collaboration with a telecom provider in China, an IBM team has prototyped an ITS system where one of the basic operations is to recreate vehicle trajectories from mobile phone GPS data. The problem of trajectory mapping [36] requires *reconstructing* the route taken by a vehicle from a sequence of discrete positions captured by the distributed sensors.

A formulation of the trajectory mapping problem reduces to resolving the shortest path problem with additional constraints. Specifically, in an undirected graph, a vertex denotes a physical location in the real world and an edge connecting two vertices denotes a road segment. The weight associated with an edge represents the length of the corresponding road segment. Considering an initial vertex v_i and a destination vertex v_d, as well as a sequence of observed vertices v_0, $v_1 \ldots$, v_n, the shortest path from v_i to v_d must satisfy the following two constraints: (1) the shortest path must include the observed vertices v_0, $v_1 \ldots$, v_n, in sequential order; (2) the total number of vertices in the path is less than or equal to a predefined threshold l, allowing for controlling how accurate the recovered trajectory is.

Trajectory mapping schema:

```
CREATE TABLE map (src integer, dst integer, path string,
                  length integer, PRIMARY KEY (src, dst))
```

Trajectory mapping SQL query:

SELECT path, length FROM map
  WHERE src = $\underline{v_i}$ AND dst = $\underline{v_d}$ AND path LIKE $\underline{\text{pattern}}$
  ORDER BY length

Trajectory mapping query plan:

$$\begin{tikzcd} \arrow[r, "\text{map}"] & \sigma \arrow[r] & \pi \arrow[r] & \text{Sort} \end{tikzcd}$$

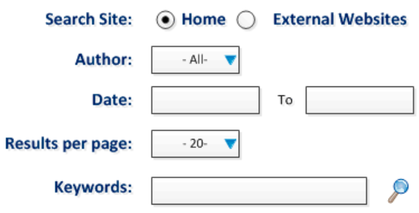
Figure 2: Original trajectory mapping query.

In our application, a typical road network represented as a graph contains tens of thousands of vertices and reconstructing trajectories on-the-fly is not usually feasible. To accelerate such computations, our application employs a path-based index data structure [30] to speed up the reconstruction of trajectories. This index data structure is very stable as the road network does not change frequently. In spite of that, this index was originally stored in a relational database and queries of the type seen in Figure 2 are issued by the application. The underlined $\underline{v_i}$, $\underline{v_d}$, and $\underline{\texttt{pattern}}$ in the SQL query are the parameters to the query that change from call to call, and the underlined $\underline{\sigma}$ in the query plan is the select operator that uses the parameters. For example, the parameters could look for paths from initial vertex $v_i = 17$ to destination vertex $v_d = 8$ that go through vertices $v_0 = 20$ and $v_1 = 5$ by using pattern = "%:20:%:5:%". The result of such queries are used to visualize road conditions as seen in Figure 1[1].

These queries account for a substantial portion of the ITS application computational load (above 90%) as this type of query is used heavily for updating the visualization interface. A quantitative analysis of the relational database implementation and other alternatives are presented in Section 6.

3.2 Market Intelligence Portal

The Market Intelligence Portal (MIP) [33] is an information system for automatically collecting market information from various sources on a continuous basis. Data sources include web sites, distributed file systems, and mail servers. Once all the data has been collected, it is stored in a relational database server. The repository is then mined and post-processed and the resulting information can be mapped onto predefined taxonomies. Searching and browsing services on these results are provided to end users as a means to, for example, understand business trends and information concerning competitors. MIP has been deployed by a customer in China for coalescing and distilling information from multiple information sources.



Search Site:	[✓ Home] External Websites
Author:	- All-
Date:	To
Results per page:	- 20-
Keywords:	

Figure 3: One of MIP's query interface (translated from Chinese)

In addition to post-processed information, the database supporting the portal contains information on a large collection of web pages, including the full page content and basic page attributes (e.g., page author and page snapshot timestamp). Figure 3 shows the user interface[2] for one of the bottleneck queries in the system. Figure 4 shows the SQL formulation for the query we experimented with, underlining the changing parameter $\underline{\texttt{idList}}$. In general, the queries generated by the portal typically go against a *stable* snapshot of the database tables. The system is continuously ingesting additional crawled data and updating the categorized information, but users tolerate slightly out-of-date query results.

Market intelligence portal schemas:

```
CREATE TABLE webpage (id long primary key, root_site_id long,
                      language string, title string,
                      content string)
CREATE TABLE rootsite ( id long primary key, summary string)
```

Market intelligence portal SQL query:

```
SELECT webpage.*, rootsite.* FROM webpage AND rootsite
       WHERE webpage.language = 'en'
        AND webpage.root_site_id = rootsite.id
        AND webpage.id IN idList
```

Market intelligence portal query plan:

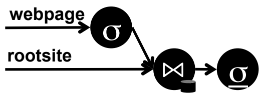


Figure 4: Original market intelligence portal query.

We collected the logs from the portal and observed that over long periods of time, this type of query accounts for about 55% of the overall processing time. Again, in Section 6, we provide a quantitative study of the original implementation using a relational database and also of the alternative configurations we have experimented with.

3.3 Spam Short Message Filtering

Short messaging or *texting* is a popular service provided by cell phone companies worldwide. With the sharp increase in the number of mobile phone users, in particular, in emerging economies such as China[3], spamming has become a concern for service providers and users alike.

Traditional anti-spam methods such as scanning the message content for keywords or, alternatively, allocating a predefined quota on the number of short messages sent per person daily usually do not provide satisfactory results, as both have clear accuracy and usability shortcomings. Facing these challenges, our team, in collaboration with a telecom operator in China, developed a new application called Spam Short Message Filtering (SSMF) using some of the techniques described earlier in the literature [6].

In essence, the application distinguishes spam from regular messages according to personal relationships between callers and callees as represented by a social network graph. In this graph, a vertex v_i denotes a mobile phone user and an edge $e_{i,j}$ between v_i and another vertex v_j (representing another user) indicates that the two users i and j *know* each other, as they have called one another in the past. The basic assumption behind such model is that spam messages are usually sent out to a large number of randomly selected targets.

In a typical configuration, the SSMF application manages a social network graph with several tens of millions of vertices. An important aspect of this dataset is that the social network evolves slowly and the spam detection technique used in the application is, to a great degree, insensitive to slight changes in the graph. This implies that scoring an incoming message against this model does not require a completely accurate graph.

[1]We show a visualization on top of a road network in the US due to a confidentiality agreement with our customer in China.

[2]The original interface is in Chinese. We show a translation to English made by our team.

[3]In 2009, the number of cell phone users in China reached 650 million [37].

Spam short message filtering schemas:

```
CREATE TABLE graph (src unsigned long primary key,
                    dst unsigned long)
```

Spam short message filtering SQL query:

```
SELECT COUNT(*) FROM graph
 WHERE src IN { SELECT dst FROM graph WHERE src=srcid }
    AND dst IN { SELECT dst FROM graph WHERE src=srcid }
```

Spam short message filtering query plan:

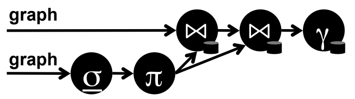


Figure 5: The original SSMF relational query.

From profiling runs with the original application, we established that queries for assessing inter-user connectivity in social network graph account for approximately 90% of the total execution time in determining whether a message is legitimate or not. Figure 2 depicts the specific SQL formulation, underlining the parameter <u>srcid</u>. The inner query SELECT dst FROM graph WHERE src=<u>srcid</u> returns the set of neighbors of the sender srcid of the message. If we denote this set as N(srcid), then the outer query ... FROM graph WHERE src IN N(srcid) AND dst IN N(srcid) finds pairs src/dst of neighbors that are themselves connected by an edge in the graph. In other words, the three nodes $\{\mathsf{srcid}, \mathsf{src}, \mathsf{dst}\}$ form a triangle. A message looks legitimate if the count of such common acquaintances is high.

Again, in Section 6, we provide a more quantitative characterization of the challenges faced in the context of the original implementation as well as the improvements we were able to obtain.

4. BACKGROUND

This section briefly describes the fundamental technologies used in the present work: the ODBC interface, the System S stream processing platform, and its programming language, SPADE. Our aim here is primarily to provide enough information to the reader to understand how we architected our framework (which we describe in Section 5), as opposed to be comprehensive. Each of these technologies has substantial literature that describe them in detail.

4.1 ODBC-based Applications

Embedding database calls in the source code is the most prevalent way of writing applications that rely on relational database-managed state such as the application described in Section 3. Most databases provide a call level interface (CLI) to application writers. Nevertheless, it is common for application writers to want to isolate an application from the database the application employs. This approach is usually preferred because it affords a degree of independence from database vendors, allowing the database server to be replaced, if necessary.

Typically, this isolation is realized by employing ODBC interfaces as opposed to the native CLI. The Open Database Connectivity interface (ODBC) provides APIs for using database servers. The designers of ODBC aimed to make it independent of programming languages, database systems, and operating systems. Most database vendors make ODBC interfaces available with their servers. Drivers exist for enterprise servers such as Oracle, DB2, Microsoft SQL Server, Sybase, Pervasive SQL, MySQL, PostgreSQL, and desktop database products such as FileMaker and Microsoft Access, running on many operating systems, ranging from Windows to several flavors of Unix.

4.2 System S and the Spade Language

Emerging streaming workloads and applications gave rise to new data management architectures as well as new principles for application development and evaluation. Several academic and commercial frameworks have been put in place for supporting these workloads. System S [2, 27, 38] is a stream processing middleware from IBM Research. System S supports structured as well as unstructured data stream processing and the execution of multiple applications from a community of users, simultaneously. These applications can be scaled to a large number of compute nodes and can interact at runtime through stream importing and exporting mechanisms. System S applications take the form of dataflow processing graphs. A flow graph consists of a set of PEs (processing elements, i.e., execution containers for the application logic stated as a collection of operators) connected by streams, where each stream has a fixed schema and carries a series of tuples. The *operators* hosted by PEs implement stream analytics and can be distributed on several compute nodes. System S provides a multiplicity of services, such as fault tolerance mechanisms [26], scheduling and placement mechanisms [19], distributed job management, storage services, and security.

SPADE [20, 23] is the programming language for System S. The SPADE tooling includes a rapid application development environment, as well as visualization and debugging tools [12, 18]. The language can be used to compose parallel and distributed stream processing applications, in the form of operator-based dataflow graphs. The language makes available several operator toolkits, including a *stream relational* toolkit that implements relational algebra operations in the streaming context; an *edge adapter* toolkit comprising operators for ingesting data from external sources as well as publishing results to external consumers, such as network sockets, databases, file systems, as well as to proprietary middleware platforms. A distinctive feature of the SPADE language is its extensibility. New type-generic, configurable, and reusable operators can be added, enabling third parties to create application or domain-specific toolkits of operators.

5. THE APPLICATION ACCELERATION FRAMEWORK

Accelerating applications of the type described in Section 3 is fundamental in the sense that they become able to cope with additional workloads and thus meet the business requirements associated with a growing user base. Moreover, it is fundamental that an acceleration framework provides the means for the application to be inherent *scalable*. In other words, the data processing infrastructure supporting the data-intensive part of the application must be able to grow and effectively use additional resources, when these become available.

The framework described in this section meets these criteria. First, it transparently replaces the relational database server component with an equivalent streaming application.

Second, the streaming application is built such that, through a modification of the dataset layout and the corresponding distribution of the dataset over additional machines, it is scalable and capable of exploiting additional processing and storage resources when they are made available.

While there is a family of applications that can benefit from our framework, the approach clearly is not applicable to all applications that make use of relational databases. Hence, it is useful to recap the properties of the applications we target:

- **Streaming paradigm:** The application must have a streaming nature, i.e., some portion of computation performed by the application is triggered by an external and continuous data source.
- **Infrequently changing and read-only datasets:** In this application, the query (or queries) whose cost dominates the application response time requires accessing a dataset that is read-only and updated infrequently.
- **Partitionable datasets:** The read-only dataset can be declustered and distributed across multiple backend servers. Such distribution has to take into consideration the query or queries submitted by the application and how different queries can be routed to different backend servers based on simple properties such as a unique identifier (e.g., a customer id).

In the rest of this section, we look into the steps required by our framework to accelerate applications. We break these steps into a compile-time phase and a runtime phase.

5.1 Transforming the original application

Our framework employs two basic strategies for accelerating the original application. First, it identifies the application's *bottleneck* queries, i.e., the queries responsible for the highest portion of the processing cost in producing the application results (e.g., asserting that a particular text message is not spam in the spam filtering application). Second, it relies on computation and dataset distribution to efficiently leverage computing and I/O resources provided by one or more backend servers, which run a distributed version of the bottleneck queries against in-core and partitioned datasets, representing the original database tables.

Employing the strategies we outlined above requires performing four steps: (1) profiling the application and identifying the queries that account for the bulk of the application response time; (2) applying modifications to the original application; (3) generating a streaming application for the targeted queries from the original application; (4) distributing the datasets onto the backend servers that will host the query processing streaming application. We now describe these steps in a more detailed way.

5.1.1 Profiling the application

The application to be accelerated must be initially run in profiling mode. In this step, we collect timing information regarding each ODBC call made by the application. The timing information is post-processed and the results are aggregated on a per-query basis (i.e., the different types of queries an application may issue against the database), and, finally, ranked based on the amount of time that each type of query demands.

The profiling information is used in the next step to decide which queries will be converted into a System S streaming application.

5.1.2 Modifying the application

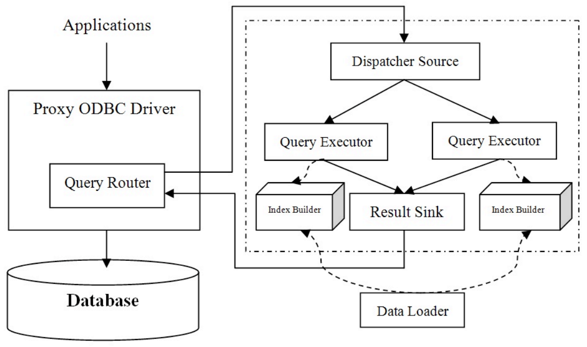


Figure 6: Architecture.

Figure 6 illustrates the core modification to the original application source code, which consists of intercepting all ODBC calls and proxying them through a replacement ODBC driver we implemented, the *proxy ODBC driver*. This ODBC[4] driver has the capability of routing queries to either the relational database server or, alternatively, to a System S streaming support application (described in Section 5.1.3) that acts as a customized query engine for a specific type of query.

As we mentioned earlier, the interaction between the application and the database server is carried out via ODBC calls. Hence, the first step in modifying the original application consists of two tasks. First, all ODBC calls must be identified and replaced with drop-in replacement calls to our proxy ODBC driver. Second, the specific statements for the SQL queries must be extracted to be used in a code generation phase we describe later.

The list of extracted SQL queries is then manually analyzed under the light of the profiling information that was previously collected as described in Section 5.1.1. At this stage, the developer has the option of tagging one or more queries for replacement by the proxy ODBC driver, according to their profiled cost. All of the *tagged* queries are then converted into native System S streaming applications as we will describe shortly. The catalog of converted queries is kept by the proxy ODBC driver for query routing decisions to be made at runtime as will be described in Section 5.2.

5.1.3 Generating the support streaming application

The support streaming application accelerates an SQL query (that the user tagged as expensive) by running it on System S. This section describes the step of generating the streaming application. This step is currently carried out in a semi-automatic fashion.[5] Our design is driven by the twin goals of improving *scalability* by utilizing many shared-nothing hosts in a cluster, while keeping *fidelity*, i.e., staying faithful to the semantics of the original SQL query.

[4]Note that, while not implemented, supporting JDBC-based applications is also possible with the same general mechanism outlined in this work.

[5]We are currently working on the mechanism to fully automate the application generation.

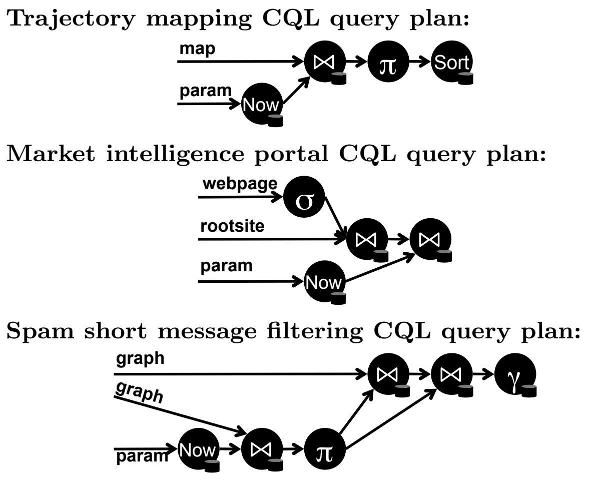

Trajectory mapping CQL query plan:
map
param
Now
⋈
π
Sort
Market intelligence portal CQL query plan:
webpage
σ
rootsite
param
Now
⋈
⋈
Spam short message filtering CQL query plan:
graph
graph
param
Now
⋈
π
⋈
⋈
γ

Figure 7: Transformation from SQL to CQL.

5.1.3.1 From SQL to CQL.

CQL, the continuous query language, is a StreamSQL dialect [4, 5]. We chose CQL as an intermediate step for our accelerator framework because it has a formal denotational semantics that reuses the classical relational operators with their usual meaning, thus making it easy to remain faithful to SQL [5]. The intuition behind the transformation from SQL to CQL is to go from a stream of SQL queries to a continuously running CQL query by turning the parameters that change between successive invocations of the SQL query into an input stream for the CQL query. To make this more concrete, consider an SQL query such as the following:

SELECT *attrs* FROM *relations* WHERE *cond*($\underline{param}$)

In this query, the underlined part is a parameter. To obtain a CQL query, we turn this parameter into an input stream:

SELECT *attrs* FROM *relations*, Now($\underline{param}$) WHERE *cond*

The Now operator in this transformed query is a window operator that turns a stream into a relation. This is necessary because the join operator (⋈) expects two relations, since CQL reuses standard relational operators for fidelity. Figure 7 shows the CQL plans that result from applying the transformation to each of the tagged queries in the example applications from Figures 2, 4, and 5. In each case, the CQL plan replaces the underlined select operator ($\underline{\sigma}$) by a join operator (⋈) with an input from a parameter stream.

5.1.3.2 Distributing CQL.

The step of transforming from SQL to CQL kept fidelity, but did not yet achieve scalability, because the execution engine described by the CQL paper is not distributed [4]. The original CQL execution engine relies on two assumptions that impede distribution. It assumes shared memory, because synopses (operator-local data) and streams hold pointers to shared actual tuple objects. And it assumes a centralized scheduler, because the operators must work in topological order to guarantee input availability.

In prior work, we presented an alternative semantics for CQL that avoids these two assumptions for the purpose of being more amenable to distribution [31]. We achieve that by translating from CQL to our core stream calculus Brooklet, which has a small-step operational semantics that models distributed implementations. Our paper on Brooklet provides a proof that our alternative semantics keep fidelity by still using the same relational operators as CQL.

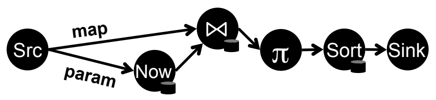


Figure 8: CQL query+source+sink = SPADE graph.

Compared to the Brooklet-based semantics in our previous paper, the present work speeds up the implementation through two additional steps. First, we use *incremental messages*: we avoid sending the entire contents of each relation at each timestamp, by sending only the changes in the relation compared to the previous timestamp. This is similar to the implementation described by the designers of CQL [4].

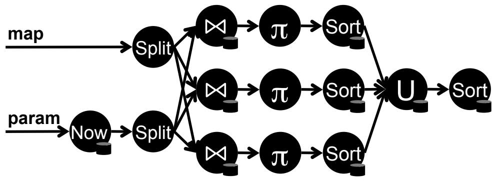
Second, we use *phantom messages*: we avoid sending empty messages when no changes occur along one path. In the original semantics, CQL sends one message on every edge in the query plan at every timestamp. This achieves determinism, because confluence operators (operators with $>$ 1 input edges) wait for input messages with matching timestamps. However, thanks to the "infrequently changing and read-only datasets" property mentioned earlier, data on some edges rarely changes, and thus the message is empty most of the time. Sending those empty messages is wasteful, but in their absence, confluence operators need some other way of determining that all inputs are up to the latest timestamp. We call an empty message that is not sent a *phantom message*, and indicate it by setting a *phantom bit* on a message on the sibling path. When a message with phantom bit arrives at a confluence operator, the operator proceeds without waiting for another message for that timestamp. Details of this idea are omitted for space reasons.

5.1.3.3 Distributed CQL on System S.

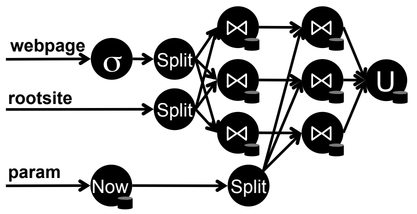
Figure 8 shows the flowgraph of the System S application for the trajectory mapping bottleneck query. Each vertex in the CQL query plan turns into an operator in a SPADE application flowgraph. In addition, the SPADE graph has two extra operators, one source and one sink. All operators (source, sink, and "regular" CQL operators) are generated from templates through a synthesis harness, as mentioned in Section 4.2. For example, there is a template for a join operator (⋈), and the synthesis harness instantiates different optimized versions of the template for different schemas and join conditions. One benefit that CQL lends to our acceleration framework is that we can use the same operators and the same SPADE application for both normal execution and for index building.

During *normal execution*, the proxy ODBC driver routes bottleneck queries to the SPADE application. Those queries are received by an input socket of the source operator. The source operator tags the parameters with consecutive timestamps starting at 1, and sends them on the param stream shown in Figure 8. From there, the data is processed by the CQL query plan. The query plan naturally exploits pipeline parallelism, and uses the timestamped CQL semantics to assemble results correctly at confluence operators. The SPADE compiler may fuse some operators to reduce communication overhead [19], depending on how the logical query plan is mapped to physical machines, and depending on profiling information. Finally, when results arrive at the sink opera-

Trajectory mapping partitioned query plan:



Market intelligence portal partitioned query plan:



Spam short message filt. partitioned query plan:

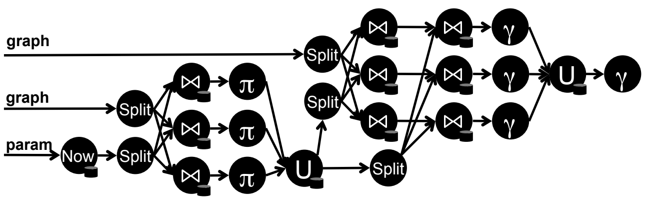


Figure 9: Query plans after partitioning.

tor, the sink relays them back to the proxy ODBC driver.

During *index building*, the source operator bulk-loads relations from the database, tags them with timestamp 0, and sends them along the appropriate edges. For example, in Figure 8, the source operator loads the map relation and sends it on the map edge. The data from the relations flows through the CQL query plan, eventually coming to rest in the operator-local state area, which is called synopses in CQL terminology [4]. Besides getting the data in the right place and the right representation, index-building sometimes even performs some pre-computation. For example, consider the CQL query plan for the market intelligence portal in Figure 7. At timestamp 0, the webpage relation is first filtered by a select operator, then joined with the rootsite relation, and finally comes to rest in the synopsis of the final join operator, which will later combine it with the parameters during normal execution.

The *phantom messages* are key to making this dual use of the SPADE application for both index building and normal execution efficient. At timestamp 0, the param path through the graph lies idle, by being treated as phantom messages while the relation paths are active. At timestamps ≥ 1, the relation paths through the graph lie idle, by being treated as phantom messages while the param path is active. However, all paths through the graph use the same operators, with symmetric algorithms and data structures. This reuse reduces engineering effort and increases our confidence in the fidelity of the accelerator: both index building and normal execution follow CQL semantics, which, in turn, is faithful to the meaning of the classical relational operators.

5.1.4 *Partitioning and distributing the dataset*

Section 5.1.3 showed how to turn a bottleneck SQL query into a streaming application for System S, via the intermediate steps of creating a CQL query and its physical distribution plan. That transformation already provided pipeline parallelism, but the benefits of pipeline parallelism are limited by the depth of the query plan. To increase the available parallelism, we partition the data. Our partitioning technique is fairly standard [15], and is enabled by the previous step of distributing CQL on System S. We have not yet automated the partitioning; instead, we transform the stream graphs by hand for our case study applications. Besides the aforementioned benefit of increasing available parallelism, partitioning yields the additional benefit of reducing the amount of data on each node in the cluster. This allows us to limit the space requirements on each node to fit comfortably in main memory, yielding speedups over disk-based databases.

The "partitionable datasets" requirement mentioned earlier mandates *key attributes* on the input relations and parameters. Our partitioning uses a hash-split operator that routes data to different subgraphs. As before, index building and normal execution are treated no differently, both use the same split and other operators for fidelity and to keep our acceleration framework simple.

Figure 9 shows the partitioned query plans for $N = 3$ nodes. Configurations for other values of N follow a similar pattern. The Split operator always partitions by the primary key attributes from the SQL queries in Figures 2, 4, and 5. For example, the Split operator in the trajectory mapping application partitions by src and dst, and the join operator performs equality comparisons on these before filtering with the LIKE predicate. As for the union ($\cup$) operators, the trajectory mapping application performs one Sort still in the split subgraph, exploiting the cluster nodes better, but requires a final Sort after the union. Similarly, the spam short message filtering application pre-aggregates the count (γ) in the split subgraph and finalizes it after reunion. The spam short message filtering application has a nested subquery, which uses different key attributes. Therefore, the subquery and the main query turn into two separate split/union subgraphs.

Overall, the load balancing of our partitioning scheme depends on how well hashing spreads the workload. The less skew in the hashed key attributes, the better the load balancing, and the better the overall performance. At the time of this writing, we use manually written hash functions.

5.2 Running the streaming application

Once the original application has undergone the modifications summarized in Section 5.1, the new application can be run normally as the changes are completely transparent to the original application.

This transparency is accomplished by making use of a proxy ODBC driver as we stated earlier. The new application receives the data to be processed and goes through its normal processing. Based on the incoming workload, a query statement is prepared as usual (using the `SQLPrepare` ODBC interface) and executed (using the `SQLExecute` ODBC interface). Once the query is received by the ODBC driver, it makes a routing decision pertaining to where the query is to be executed. A query may be executed by the original relational database server or via the System S support application. This is a simple decision. The proxy ODBC driver has a catalog of which query types should be directed to the database and which should not. For queries shipped to the database server, a simple call is made through the original ODBC driver. In other words, our ODBC driver acts purely as a proxy. For the queries shipped to the System S support application, the variable parameters are removed from the query specification and are directly sent to the support streaming application.

The support streaming application has a *Dispatcher Source* that receives these parameters and initiates the query processing. The *Dispatcher Source* routes the query to the *Query Executor* subgraph. Once the *Query Executor* computes the result for the query, it submits the results to the *Result Sink* operator, which, in turn, returns the results to the proxy ODBC driver, making the results available to the application for further processing as required.

Our framework has been designed to include the capability of hot-swapping the support streaming application on-the-fly, when the datasets employed for computing the query results are updated. In other words, a new version of the System S support application loaded with the new version of the read-only datasets can be brought up in tandem with the original version. Once the new one becomes fully operational, the old version is terminated. Our replacement ODBC driver can detect the change and start routing the queries to the new version, seamlessly. This capability is still experimental and available only in prototype form.

6. EXPERIMENTS

The design of our database acceleration framework aimed at improving the performance of certain types of applications that must process a stream of incoming queries. To assess how effective our proposed framework is, we employed the applications described in Section 3. We empirically studied them considering two key metrics: *query latency* and *query throughput*.

Query latency is important because it gives an indication of how long it takes for a query engine, in this case either a relational database server or the equivalent SPADE streaming application produced by the acceleration framework, to compute the query results on behalf of the front-end application. Query throughput is important because, ultimately, the goal is to scale up the original application to accommodate higher workloads demanded by a production deployment. In other words, as the load increases (e.g., as more mobile phone users are added by the telecom provider using the SSMF application) and as more computational resources are made available to the application, the goal is to keep the number of queries answered per unit of time constant, ideally.

Our study was conducted in a Linux cluster, using a stock distribution of System S. Our cluster has four 2.66 GHz Intel Core 2 6700-based nodes, each with 12 GB memory and a 160 GB hard disk, running RedHat Enterprise 4.0.

For each of the case study applications we extracted the bottleneck query (described in Section 3), identified by the profiling phase discussed in Section 5. We then developed 5 versions of a query execution mechanism for computing this query. The query execution mechanism consists of a *query engine* and the dataset needed for executing the bottleneck query. The query engines we employed included the original relational database used by the application, which was IBM DB2 in all cases, as a baseline as well as the following alternative configurations:

- **SolidDB-RW**: The SolidDB-RW configuration employs IBM's SolidDB. SolidDB is an in-core relational database [24]. It can be configured both in read-write mode (when queries can also update tables) or in read-only mode (when updates are not allowed). The SolidDB-RW configuration employs the database server in read-write mode, even though that was not needed by our case study applications.

- **SolidDB-RO**: The SolidDB-RO configuration was setup such that the database was capable of executing only read-only queries.

- **CQL on System S**: The CQL on System S configuration corresponds to the application as produced by the code generation mechanism described in Section 5.1.3.

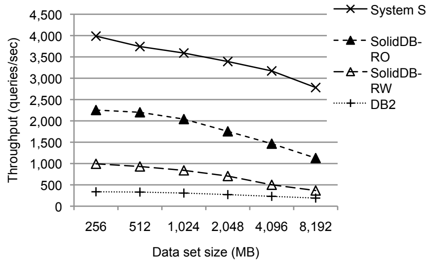
- **Manual System S**: The Manual System S configuration corresponds to the application generated by our framework, but with additional manual optimizations. The manual tweaks were made only for the SSMF application. In this case, the code generated by the CQL-translation phase is further optimized as is the layout for the data. Locating the neighbors to a vertex can benefit from indexing, reducing the complexity of the search at runtime. We omit the details for space reasons; our methodology is similar to Schank and Wagner's "edge-iterator" algorithm [29]. Indexing is done offline and the modified runtime algorithm assumes the presence of the index. We have not attempted to further optimize the auto-generated code for the MIP and TM applications.

The set of query workloads we employed were generated by randomizing the application traces we had from earlier field deployments. In all experimental configurations we used as much of the real data for each application as we could fit in memory. In other words, we truncated the datasets based on time intervals, constrained by how much in the past we could go and still keep the data in main memory.

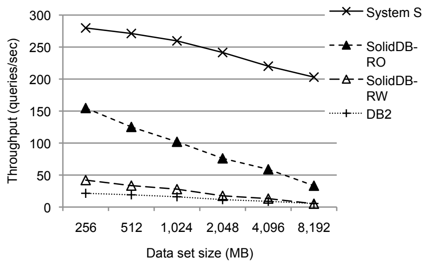
6.1 Varying the dataset size

In the first set of experiments, we looked at each application's bottleneck query from the standpoint of each query engine's sensitivity to the dataset size. In this case, all applications were run on a single node.

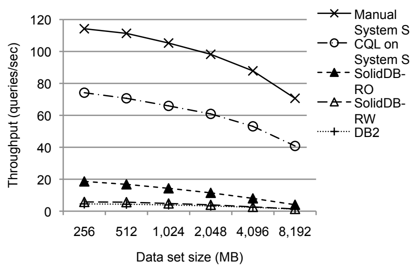
The first set of experimental results are depicted by Figure 10 and focused on query throughput. As expected, all query engines exhibited a decrease in throughput as the datasets increased in size. Not surprisingly, the baseline DB2 configuration was the slowest as it experiences all of the transactional costs incurred by a disk-based relational server. The in-core capabilities provided by SolidDB are responsible for a great deal of improvement in performance, with and average of a 2-fold to 5-fold increase in throughput for the TM and MIP applications. Even more substantial improvements were observed for the SSMF application with the SolidDB-RW and SolidDB-RO configurations, respectively, irrespective of the dataset size. On top of such gains, our CQL-translated streaming application was able to attain between 1.5 and 2-fold additional improvement in query throughput when compared to SolidDB-RO. These improvements can be attributed to the application-tailored code produced by the SPADE compiler (in contrast with the generic query engines provided by the relational databases) as the CQL query is translated into a System S application.



(a) Trajectory mapping.

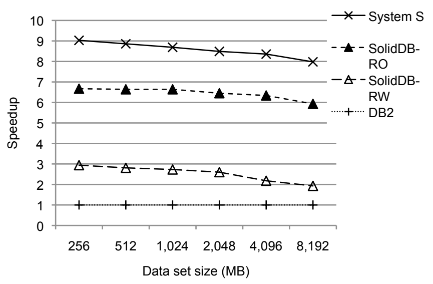


(b) Market intelligence portal.

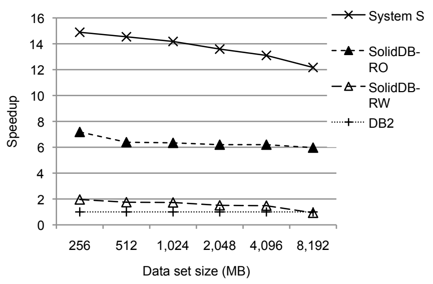


(c) Spam short message filtering.

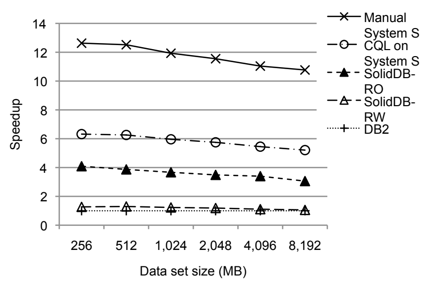
Figure 10: Throughput vs. data set size for bottleneck queries, on one cluster node.



(a) Trajectory mapping.



(b) Market intelligence portal.



(c) Spam short message filtering.

Figure 11: Normalized latency vs. data set size for bottleneck queries, on one cluster node, relative to DB2.

Our second set of experimental results, depicted by Figure 11, focus on latency. In particular, we looked at the speedup provided by the alternative query engines when compared to the one relying on DB2. Surprisingly, sharper improvements were observed in query latency. Both SolidDB-based configurations provide substantial speedups in most cases, up to a factor of 7 (for the TM application bottleneck query when employing the SolidDB-RO configuration). The speedups tend to decrease with the dataset size as the cost of memory accesses increases. In all cases, the application produced as a result of the CQL translation step delivers speedups varying from an 8-fold improvement to a 15-fold improvement.

The SSMF query clearly benefits from the manually inserted optimizations, with an extra 50% increase in throughput and almost double of the speedup provided by the automated translation, irrespective of the dataset size.

6.2 Varying the number of cluster nodes

In the next set of experiments, we looked at each application's bottleneck query from the standpoint of the sensitivity of each query engine configuration to distributed execution as we increased the number of cluster nodes (and dataset partitions) we employed. In this case, each data partition is hosted on a different node and we made use of the query distribution strategy described in Section 5 to load balance the stream of incoming queries. Again, we examined the impact on query throughput and latency.

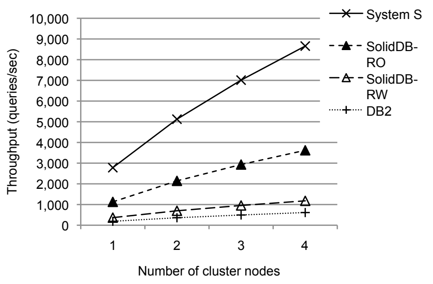
As before, the first set of experiments depicted by Figure 12 focuses on throughput. Once again, we see that all the configurations are reasonably well-behaved, scaling almost linearly with the increase of computational resources. Nevertheless, the relative difference between the different configurations is noticeable. The general trend is that SolidDB-based configurations outperform DB2, but are also outperformed by the CQL-translated Spade application. In some cases, the improvement translates into triple the throughput of the SolidDB-RO configuration (specifically, the 4-node configuration for the MIP application). As before, we see that the manual tweaks deliver superior results for the SSMF application, demonstrating that more in-depth compiler-based analysis in the CQL translation might be worthwhile.

6.3 Studying the effects of data skew

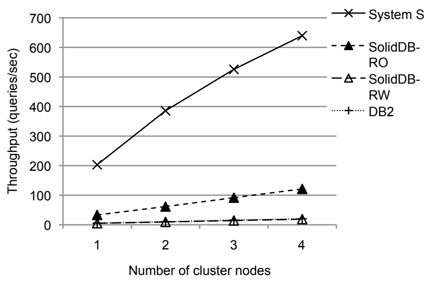
The experiments depicted by Figure 13 focus on data skew and its effect on throughput when the applications are run in distributed mode. In particular, we were interested in understanding how data skew affects the load balancing in distributing queries to different nodes. For these experiments, we rigged the hash functions to produce skew. The x-axis gives the amount of skew as the percentage of data that is stored on the most heavily loaded node. For example, with 4 nodes, 25% on the most heavily loaded node corresponds to an even data distribution (no skew), whereas 70% on the most heavily loaded node leaves only 10% each for the other three nodes (heavy skew). As expected, the benefits of data parallelism diminish with increasing skew. For example, the performance of the 4-node configuration drops approximately to the performance of the 3-node configuration when skew increases from 25% to 33%, since throughput is now bounded by a node with the same load. To conclude, partitioning helps most when skew is small (as expected), highlighting the importance of using hash functions that provide good workload spreading. All three of our case study applications have negligible skew.

6.4 Beyond the bottleneck queries

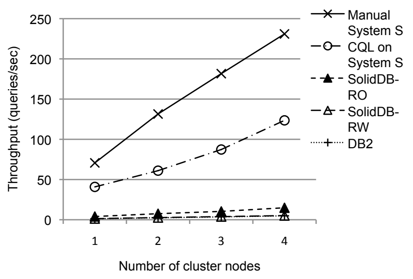
All the previous experiments focused on executing the bottleneck queries in isolation. In the experiments depicted



(a) Trajectory mapping.

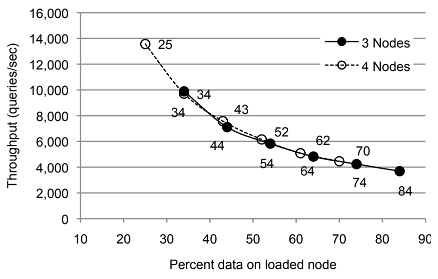


(b) Market intelligence portal.

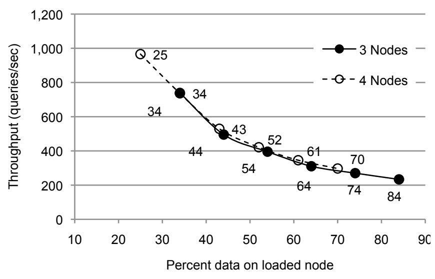


(c) Spam short message filtering.

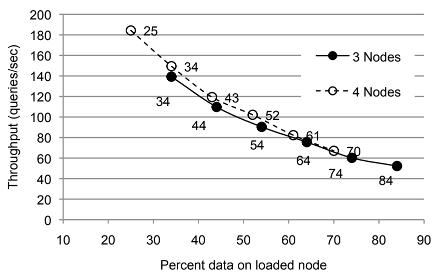
Figure 12: Throughput vs. number of nodes for bottleneck queries, with 8GB data set for each application.



(a) Trajectory mapping.



(b) Market intelligence portal.



(c) Spam short message filtering.

Figure 13: Throughput vs. data skew for bottleneck queries, with 8GB data set for each application.

by Figure 14, we ran the full application, but still measured the performance of the bottleneck queries as a proxy for the application performance. The intention was to assess the effect of running the full application on the throughput and latency for those queries, as computational resources are diverted to additional processing.

Interestingly, the results are generally consistent with those seen in Figure 10, as the bottleneck queries do account for the bulk of each application's computational budget. For this reason, we omit the latency results as they also track what we had seen in Figure 11.

6.5 Discussion

Our empirical evaluation demonstrates that the code generation approach employed by our CQL translator, coupled with our ODBC-driver replacement, is a viable alternative to improving the performance and scaling up this class of applications that processes a stream of queries against read-only and infrequently updated datasets.

Despite being very promising, we consider our results still a work-in-progress. As seen from the relative difference between the automated translation and the manually tweaked configuration for the SSMF application, our CQL translator still has considerable room for improvement. Furthermore, we consider our tooling as not being at production-level yet. For example, additional improvements will have to be made to aid in the automatic identification of queries that can be efficiently translated. More work is also needed to create mechanisms and tooling for automating dataset partitioning and workload distribution.

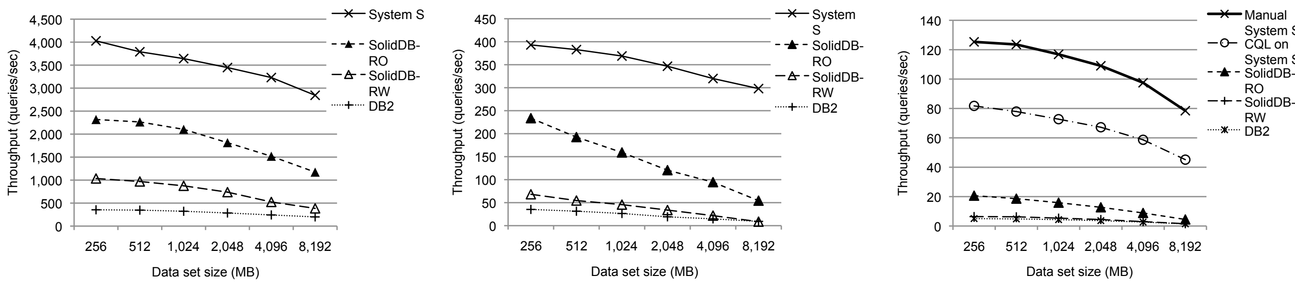
7. RELATED WORK

Database acceleration and, in general, query optimization has been a prolific area of research. Multiple surveys over the last two decades have summarized the state of the art techniques in this area. A non-exhaustive list includes the work done by Graefe [21], Ioannidis [25], and Chaudhuri [9]. Substantial improvements in commercial relational database systems have resulted from the pioneering work described in these surveys.

For several applications that do not require the guarantees provided by regular relational databases, both researchers and vendors have looked at in-memory database implementations [16, 22] to speed up the execution of queries. From TimesTen [34] (later acquired by Oracle) to Solid Information Technology [24] (later acquired by IBM), several products are available in the market. There has also been substantial work on parallel database technologies [14, 15] and database federation frameworks [28], which provide improvements in raw database performance and integration capabilities. There has also been work on read-only transactions in databases [17].

There has been work on novel, distributed data processing frameworks, focusing on large-scale datasets. Examples include work on database servers that make use of dataset declustering techniques [8] and the now famous map-reduce paradigm [13]. Likewise, stream processing has been an active research area for over half a decade. This has led to the development of several academic and commercial platforms. STREAM [3], Borealis [1], StreamBase [32], and TelegraphCQ [7], among others, have focused on providing stream processing query languages, optimization techniques, and runtime middleware.

We believe that the core contributions in the present paper are unique in light of the earlier work, but it clearly is in the confluence of these multiple areas, leveraging the ideas from the parallel database and from the stream computing research. We think that this work presents a reasonable



(a) Trajectory mapping. (b) Market intelligence portal. (c) Spam short message filtering.

Figure 14: Throughput vs. data set size for entire application, on one cluster node.

method for *transforming* applications that deal with streaming workloads, which, in several applications, are typically implemented using traditional relational database technologies, into true streaming applications, making use of the flexibility afforded by stream processing middleware.

8. CONCLUDING REMARKS

In this work we have described an approach for modifying applications that have a streaming nature, but whose original implementations rely on traditional relational databases for the bulk of their processing. The end result is an application that leverages the capabilities of a distributed streaming middleware for computationally expensive queries, enabling the scaling up of processing in concert with increases in workload, while maintaining low latencies.

We have shown that there is an emerging class of applications that require timely data processing due to the continuous nature of their processing. The method described in this work, while not fully automated, is very effective in guiding the transformation of relational queries that are performance critical in these applications into customized distributed streaming applications. As was experimentally demonstrated in the context of several real-world applications, such transformations result in substantial performance and scalability improvements as we have shown.

We believe that the ideas proposed here are a step in evolving legacy applications to a streaming formulation where *data in motion* is processed via middleware that enables continuous query processing, while *data at rest* resides on database servers and warehouse for offline processing as well as long-lead time business intelligence applications.

9. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research, CIDR*, 2005.

[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Proceedings of the Workshop on Data Mining Standards, Services and Platforms, DM-SSP*, 2006.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, 2003.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Data Bases*, 15(2), 2006.

[5] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3), 2004.

[6] P. O. Boykin and V. P. Roychowdhury. Leveraging social networks to fight spam. *Computer*, 38(4):61–68, 2005.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference on Innovative Data Systems Research, CIDR*, 2003.

[8] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. H. Saltz. Titan: A high-performance remote sensing database. In *Proceedings of the International Conference on Data Engineering (ICDE 1997)*, pages 375–384, 1997.

[9] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS 1998)*, New York, NY, USA, 1998. ACM.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR 2003)*, 2003.

[11] K. Dasgupta, R. Singh, B. Viswanathan, and A. Joshi. Social ties and their relevance to churn in mobile telecom networks. In *Proceedings of the 2008 International Conference on Extending Database Technology (EDBT 2008)*, March 2008.

[12] W. De Pauw and H. Andrade. Visualizing large-scale streaming applications. *Information Visualization*, 8(2), 2009.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] D. Dewitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transations*

on Knowledge and Data Engineering, 2(1):44–62, 1990.

[15] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[16] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transations on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[17] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions Database Systems*, 7(2):209–234, 1982.

[18] B. Gedik, H. Andrade, A. Frenkiel, W. De Pauw, M. Pfeifer, P. Allen, N. Cohen, and K.-L. Wu. Debugging tools and strategies for distributed stream processing applications. *Software: Practice and Experience*, 39(16), 2009.

[19] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *Proceedings of the 2009 Conference on Information and Knowledge Management (CIKM 2009)*, 2009.

[20] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 2008)*, 2008.

[21] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[22] S. Graves. In-memory database systems. *Linux Journal*, 2002(101):10, 2002.

[23] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. Spade – language specification. Technical Report RC24987, IBM Research, 2009.

[24] IBM. SolidDB. `http://www-01.ibm.com/software/data/soliddb/`.

[25] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1), 1996.

[26] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu. Language level checkpointing support for stream processing applications. In *Proocedings of the 2009 International Conference on Dependable Systems and Networks (DSN 2009)*, 2009.

[27] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the International Conference on Management of Data (SIGMOD 2006)*, 2006.

[28] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: a new flavor of federated query processing for DB2. In *Proceedings of the International Conference on Management of Data (SIGMOD 2002)*, 2002.

[29] T. Schank and D. Wagner. Finding, counting, and listing all triangles in large graphs, an experimental study. In *Workshop on Experimental and Efficient Algorithms (WEA)*, 2005.

[30] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the Symposium on Principles of Database Systems (PODS 2002)*, 2002.

[31] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A unified semantics for stream processing languages. In *Proceedings of the 19th European Symposium on Programming (ESOP 2010)*, 2010.

[32] StreamBase Systems. `http://www.streambase.com/`.

[33] Z. Su, J. Jiang, T. Liu, G. T. Xie, and Y. Pan. Market Intelligence Portal: an entity-based system for managing market intelligence. *IBM Systems Journal*, 43(3), 2004.

[34] C. TimesTen Team. In-memory data management for consumer transactions the timesten approach. *SIGMOD Record*, 28(2):528–529, 1999.

[35] D. S. Turaga, O. Verscheure, J. Wong, L. Amini, G. Yocum, E. Begle, and B. Pfeifer. Online FDC control limit tuning with yield prediction using incremental decision tree learning. In *(Sematech AEC/APC 2007)*, 2007.

[36] M. R. Vieira, P. Bakalov, and V. Tsotras. Querying trajectories using flexible patterns. In *Proceedings of the Conference on Extending Database Technology (EDBT 2010)*, 2010.

[37] L. Weitao. China's mobile phone users hit 650 million. `http://www.chinadaily.com.cn/bizchina/2009-03/06/content_7547876.htm` retrieved on January 27, 2010, 2009.

[38] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proceedings of the Very Large Data Bases Conference (VLDB 2007)*, 2007.

[39] X. Zhang, H. Andrade, B. Gedik, R. King, J. Morar, S. Nathan, Y. Park, R. Pavuluri, E. Pring, R. Schnier, P. Selo, M. Spicer, and C. Venkatramani. Implementing a high-volume, low-latency market data processing system on commodity hardware using ibm middleware. In *Proceedings of the 2009 Workshop on High Performance Computational Finance (WHPCF 2009)*, 2009.