Real-Time Scheduler For Consistent Query Execution of Big Data Analytics

Shenoda Guirguis[1]
Graph DB
LinkedIn
Sunnyvale , USA
sguirguis@linkedin.com

Sabina Petride[1]
Oracle SQL Execution
Oracle
Redwood City, USA
sabina.petride@oracle.com

Abstract—**Analytical queries on big data consume a lot of resources and typically run for long time. Both resource utilization and execution time can be reduced by order of magnitude by transitioning to main memory systems, as well as by offloading part of the analytic computation to in-memory clusters of special purpose analytic engines. These systems are highly optimized for certain patterns of query execution on main memory data, and can support high level of concurrency. Trading off optimization and specialization for operational completeness, such secondary systems are not always fully fledged transactional: they hold copies of the data and rely on refreshes being coordinated from the primary. In such heterogeneous systems, it is particularly challenging to support applications with *strict consistency guarantees* requiring transaction consistent query execution. The eventually-consistency model does not fit in this setup, yet eager propagation of changes imposes a huge unnecessary overhead. In this paper, we formalize the challenge of strictly consistent query execution in hybrid (primary plus in-memory secondary) systems as a *real-time scheduling problem*, and propose a scheduler that ensures consistent query execution and minimal overhead at both primary and secondary systems. We detail the system design with a focus on the query and change propagation scheduler and its interaction with other processes, explaining the advantages of our solution over alternatives. We argue that the proposed framework is easily extendable to incorporate different customized optimization goals. We conclude with preliminary promising performance evaluation of the implemented infrastructure part of the Data Processing Unit (DPU)-based hybrid database system.**

Keywords - Big Data Analytics; Replication; Consistency; Change Propagation; Real-time Scheduling.

I. INTRODUCTION

Analytical queries on Big Data consume a lot of resources and typically run for long time [8][17]. In-memory databases speed up query performance by orders of magnitude - factor of 100x for some applications [7][9][16][21][22]. One opportunity is to use a cluster of in-memory databases, as a secondary database to speed up the performance of analytical queries [12][13][19][20]. One can consider such secondary database as a huge cache. In this setup, and in presence of data updates or changes, there is a need to ensure consistent query execution. That is, whenever a query runs against data in the secondary in-memory database, it should return the same results as when it runs against the data in the primary database. Eventually-consistent model [10] does not fit in this setup, since analytical queries are typically used to support decision making and policy changes. Therefore, analytical queries need to return accurate up-to-date results. Yet, eager propagation of changes imposes a huge unnecessary overhead. In this paper, we propose a framework that ensures consistent query execution and minimal overhead at both primary and secondary databases. This framework includes all components of a consistent query execution starting with the capture of data changes, and the propagation and deployment of changes at the secondary system. In our solution, the secondary database does not need to run the same query execution engine as the primary, which makes the framework applicable to any hybrid database, as long as there is an agreed upon data exchange format that can be understood by both systems. Further, out design allows optimizing performance for user-defined performance goals.

Our contributions can be summarized in the following points:

1) We formalize change propagation from the primary to the secondary systems and query submission on the secondary system as a *real-time scheduling* problem. While real-time scheduling is a well-known subject [2]-[6][11], its applicability to change and query propagation in heterogeneous systems is new, to the best of our knowledge. In particular, while the mechanism of capturing Data Manipulation Language (DML) activity may be common with the hybrid periodic change propagation method (of [1]), the formalization of DML activity as jobs and the definition of job granularity and job metadata are novel.

2) We explain the design of a query and DML activity scheduler that runs on the primary, its system placement in database and interaction with database processes. The scheduler is a new component of the database, and subsequently its interaction with other activities in the database is novel.

 ISBN: 978-1-61208-637-8

3) As detailed in Section V, the scheduling mechanism is efficient – it achieves its goals, formulated via the scheduling policy. Most common goals in heterogeneous systems of our focus can be formulated via scheduling policies.
The proposed scheduling system model is practical to achieve. We based our claim on a prototype implemented on heterogeneous database system and on preliminary promising performance evaluation.

The remaining of this paper is organized as follows. Section II gives the necessary background. The problem formulation and our proposed scheduler are described in Section III and Section IV, respectively. Section V details the implemented prototype in the Oracle database and the RAPID Data Processing Unit (DPU)-based in-memory accelerator [19][20], along with preliminary performance evaluation. The paper is concluded in Section VI.

II. PRELIMINARIES

The setup we are concerned with is one with two coupled-databases: a primary database and a secondary database that is optimized for analytical queries, such as the setup in [19][20] where a cluster of high-speed interconnected in-memory databases are used to boost analytical queries performance. The secondary database is not necessarily a fully-fledged Atomicity, Consistency, Isolation, and Durability (ACID) compliant database, and can be seen mainly as a query engine highly tuned to execute parts of SQL queries submitted through the primary. All activities are initiated through the primary database and execution on the secondary is transparent to the user. The user specifically loads some base tables into the target database at any point of time. When queries are submitted, the source database determines which queries or query-fragments, i.e., sub-queries, can be offloaded to the secondary database to boost query performance. Data changes, i.e., DMLs, are submitted and executed in the primary database. A change propagation protocol copies new or updated data to the secondary database, or informs the secondary about deletions.

The main advantage of such heterogeneous database systems is that they combine the full ACID compliant features of a traditional database (the primary) with the high efficiency and potentially highly distributed query execution of an accelerator (the secondary). The accelerator does not need to support all the features of the primary – the tradeoff between generalization and optimization via specialization allows highly optimized code for a narrower functionality. Fronting all operations from the primary also allows gradual support over releases of more complex features in the accelerator, and offloading more and more operations outside of the primary.

The source of truth, both in terms of data and query execution, in heterogeneous systems is the primary database. For space efficiency, only relations targeted to be queried in the accelerator are loaded into the secondary database. As all DML activity happens at the primary, the data has to be kept refreshed on the secondary after the initial load. As the secondary is not a fully-fledged transactional system, DML replication happens physically – by propagating the data that has changed from the primary to the secondary. However, there is no explicit requirement for synchronous data propagation. For instance, if DML activity happens on a relation not queried, then there is no need for the DML in the primary to "wait" until it is propagated to the secondary. The strict requirements are

- When a query executes on the secondary, it returns the same results as when it is executed on the primary. Therefore, when a query executes in the secondary all the DML activity on the relations references in the query has to be up-to-date (more specifically, up to the query system commit number *SCN*).

- Queries are offloaded to the secondary only when estimated to run faster than on the primary. The estimation should be such that applications run faster in the presence of the secondary and that the secondary is picked for execution whenever beneficial.

Therefore, in systems of our focus we have both a change propagation and a query submission problem, and they are interconnected: (1) when and how should changes be propagated to the secondary, and (2) how to choose between executing queries on the secondary system vs. on the primary, in order to ensure the above requirements?

Data replication has been utilized for decades either for availability reasons (i.e., backup and recovery) or for performance reasons (e.g., load balance). Therefore, propagation of data changes across data replicas is not a new problem. However, we argue that replication in analytic hybrid database systems of our focus poses new challenges. In heterogeneous database systems, the main purpose of the secondary database is to accelerate query execution and offload computation from the primary. In doing so, the hybrid system supports higher level of concurrency and faster query response than the primary system alone. The secondary database is neither a backup nor a replica of the primary database: queries are always submitted from the primary and within the primary a decision is taken to offload the query - if expected to run faster - to the secondary. If the primary goes down, the secondary is not accessible to users for query execution. Further, in most applications where replication is utilized for load balancing, data consistency can be sacrificed temporarily in order to maintain the performance at its best. For example, the eventually-consistent model [10] has been adopted by most key-value stores in this regard. However, in case of analytics queries, data consistency cannot be compromised or else wrong outdated conclusions may be deduced. Moreover, since the purpose of replication in data analytics is to boost performance, query performance cannot be compromised either. This poses a new challenge in efficiently and timely

propagating data changes. Therefore, existing solutions cannot be applied per se. This paper addresses these new challenges.

The spectrum of choices of how and when to propagate changes, from a source to a target database, is quite wide. At one end, changes are propagated *eagerly* as soon as they are captured. On the other end, changes are propagated *on – demand*, i.e., only when needed at query time. Alternatives include periodic change propagation – refreshes are scheduled at certain time intervals. The pros and cons of each approach make each approach suitable for certain class of applications. For example, if changes are very frequent and queries are scarce, then one can argue that the lazy approach of pushing changes on-demand is a better choice, as it amortizes the overhead of the propagation protocol. On the other hand, if query response time is critical, a proactive approach of propagating changes as soon as they are committed would be needed to minimize query response time. This, however, comes at the expense of incurring high overhead.

Typically, most systems that employ one or the other method leave it to the user or the database administrator (DBA) to choose between the methods. For instance, the secondary system can be exposed as a cache that can be configured for one or the other types of refreshes [15]. The assumption is that the user has an expectation of the workload and pattern of data changes. Switching between different methods dynamically in response to deviations from the expected usage is not typically available.

One optimization is to combine propagating changes on-demand when a query needs them, with a *periodic* change propagation approach and an eager propagation of bulk appends [1]. The goal of such a propagation scheme is to minimize the amount, or size, of data needed to be propagated at query time to minimize the query delay. It is also meant to address run-time variations in workloads of a certain pattern – sporadic append-mainly large DMLs (such as nightly bulk data ingestion) at times of low volume of queries, with regular small online transaction processing (OLTP)-type transactions. Therefore, this approach works perfect if changes are mainly scarce and small at times of high query traffic, and it ensure strict consistency – queries are not chosen for execution on the accelerator until all dependent changes are visible to the accelerator. However, even such a hybrid scheme does not answer the following questions:

- What is a reasonable or good enough time for a query to wait for dependent changes?
- How to detect the case when changes exist while the query does not need these changes? That is, when to skip vs. when to delay a certain propagation to minimize query wait time? For example, if the changes are of a later system change number (SCN) than the query SCN, then the query does not depend on these changes and therefore does not need to wait for them to be propagated. (Note that this is similar to the query/update independence analysis of [25]-[27].)
- In the presence of multiple changes and concurrent queries, in what order should changes be propagated, and in what order should queries be submitted to the secondary?
- How to efficiently adapt to deviations in the expected pattern of DML and query activity?

There is a need to precisely formalize the change propagation model in order to answer the above questions and address the new challenges in compliance with the optimization goal.

III. PROBLEM FORMALIZATION

We formalize the change propagation problem as a real - time scheduling problem. *Tasks* to schedule are (a) the data changes to propagate, and (b) the analytic queries offloaded to the target database, that is we propose a dual query and change-propagation scheduler infrastructure. The scheduler maintains as part of its metadata the dependency information between change propagation tasks and queries. This dependency is detected and maintained to ensure valid schedules. Each task is assigned a priority based on the optimization goal. Based on the dependency between tasks and their priority, the scheduler can answer the questions that arise in case of concurrent queries and multiple data changes. For example, given the priority of each query, the scheduler can prioritize which required data change to propagate first. Also, by assigning a deadline for each query-task, as we shall explain later, the scheduler can determine when it is too long to wait for updates to be propagated, and when it is not. Finally, the scheduler can adapt different scheduling policies to suit the application and performance optimization desired. For example, one policy can optimize query throughput, another can maximize number of queries executed on the target database, say to minimize energy, and a third can minimize the query wait response time, etc.

Further, the scheduler provides infrastructure for several other functionalities. For example, one opportunity is to utilize the scheduler as a resource manager which monitors workload both on source and target databases. This allows us to add load-balance functionality. Further, the DBA or the user can monitor the system and indicate at run-time a change in the scheduling policy. In a more evolved implementation of the system, the switch between policies could happen by automatic collection of system performance and usage of simple rules for picking from available scheduling policy.

A. Change Propagation: A Scheduling Problem

Definition 1: There are two types of *tasks:*

- *query-task* (read only) - In our scheduling model a query-task actually means a query-fragment

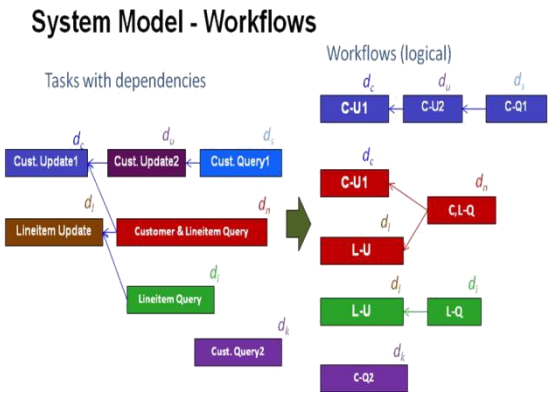

Figure 1. System Model – Workflows Example for TPC-H

that is offloaded to be executed in the secondary database to boost performance. Hence, there may be more than a one query-task per query.

- *update-task* - set of committed DMLs per object/table belonging to the same transaction.

Note that for simplicity we focus here on change propagation of committed DMLs, but the model can be naturally extended to consider uncommitted DMLs and queries submitted in a transaction after uncommitted DMLs. We note here for completeness that in the system of focus it was considered acceptable for queries running on relations in the secondary system but with uncommitted changes to be up-front executed only on the primary. Physically, update tasks can be represented in multiple formats – for instance, full post images of the changed or new rows can live in the buffer cache, on disk or in the redo log; they may also live in in-memory caches built for optimizing query execution by providing faster data access.

The dependency between tasks is defined as follows:

Definition 2: A query-task q_i and/or update-task u_j depends on update- task u_k if and only if (1) there is a common data object, and (2) u_k is executed before q_i and u_j.

We model the set of tasks as *workflows*:

Definition 3: A *workflow* is a set of tasks that has acyclic dependency relations.

A single task may then be logically present in multiple workflows. Figure 1 shows an example of seven tasks that form four workflows. The Customer update-1 task, belongs to two workflows, the first (blue) and the second (red) one. Each task (see Figure 2) has a *deadline* and a *cost*. The deadline is primarily execution driven (i.e., driven from estimated time-cost if executed in the primary database).

Definition 4: The *deadline* d_i of a query-task q_i is its estimated execution cost on the *primary* database ($c_{i,src}$). The *deadline di* of an update-task u_i is

- Infinity, if there is no query that depends on u_i
- *min*$\{d_j$ *, for all j, such that. query* q_j *depends on* $u_i\}$.

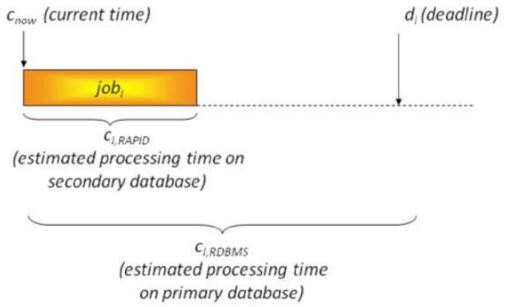

Figure 2. System Model – Individual Task

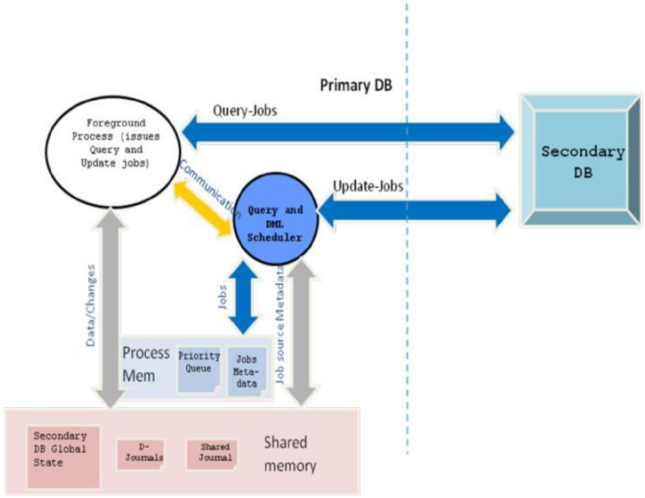

Figure 3. System Architecture

A deadline of an update-task may change, depending on arrival (addition) and departure (finish execution) of depending query-tasks. A depending query may finish before the update task it depends on, only if it is executed in the primary database, i.e., in case of fallback.

Note that the definition of cost of a query fragment is not the focus of this paper; here we rely on the fact that heterogeneous systems of our focus use a cost-based optimizer for planning a query and they are extended with cost models for execution of different query operators on the accelerator.

What is novel, however, is the usage of a query fragment cost as a deadline. In our implementation, it was a challenge to maintain, the query fragment cost as we shall discuss in Section IV. Similarly, the cost model for update tasks is not the focus of this paper. We rely on the fact that such cost models are practical, and we utilized such cost models in our prototype.

B. Change Propagation Scheduler– High Level

We assume a single propagation process with a single Propagation Priority Queue (PPQ) for the single -instance case. Figure 3 shows the System Architecture of this case. It shows that the scheduler - of queries and updates - runs in a separate process. It maintains its priority queue and tasks metadata, including dependency, in its process

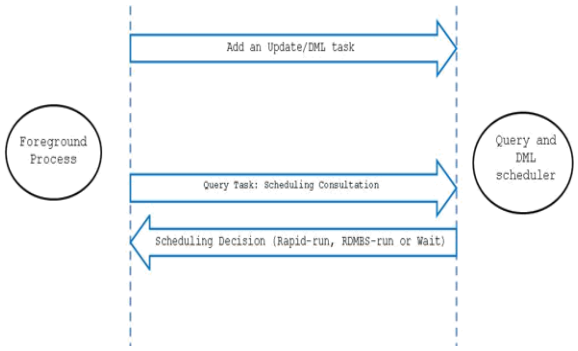

Figure 4. Communication Between Foreground And The Scheduler Process

private memory area. The query foreground processes communicate with the scheduler process to add tasks and receive scheduling decisions. This communication is depicted in Figure 4.

When a query is submitted for execution in the primary database, the query compiler determines which query fragments can be executed in the target database to boost performance. The foreground process then communicates to the scheduler each query fragment as a query-task, providing its metadata (i.e., cost, deadline, data objects, etc.).

Similarly, when a DML is submitted in the source database, updates for each data object are captured. At commit time, an update task is communicated to the scheduler process. Upon receiving a task, the scheduler updates the priority queues and tasks dependency metadata. When a DML task is due for execution, the scheduler dispatches the task shipping the updates to the secondary database. For query-tasks, on the other hand, the scheduler makes one of three decisions: (1) proceed to execute in secondary database, (2) fallback to execute on primary database, or (3) Wait. The Wait decision is basically wait for other higher priority queries, or to wait for update-task(s) this query depends on to be propagated to the secondary database first. In the following Section we provide the details of the scheduler for a single instance case.

IV. QUERIES AND DML SCHEDULER – DETAILS

In this section we detail the objectives and design for the scheduler.

A. Scheduler Objective

Our scheduler has two objectives: (1) maximize hit ratio, and (2) minimize query response time as perceived by the end-user. The *hit ratio* is the percentage of query tasks that execute within their deadlines vs. the total number of query tasks. That is the ratio of queries that meet the deadline. A task that falls back to the primary database is considered a miss. This scheduling objective also encapsulates maximizing the usage of the secondary database, which is installed to boost performance of complex analytical queries.

Existing approaches typically utilize some hybrid form of such objectives, such as MIX [2], Multiple Attribute Integration [4], and EDF & Random Hybrid [5]. All these hybrid models, however, require system parameters. An adaptive, parameter-free hybrid approach to minimize tardiness was proposed in [3]. We need a parameter-free adaptive hybrid method to maximize the hit -ratio. Shortest Remaining Processing Time (SRPT) scheduling policy is proven to provide minimal response time in case of soft-deadlines [6], i.e., when a task is allowed to run beyond its deadline. Our case however is similar to the hard-deadline one, with queries falling back to the primary.

To achieve the above two scheduling objectives, we use the invert of deadline times its secondary database cost ($p_i=1/(d_i \times c_{i,sec})$) for the task priority. Using the inverted deadline gives higher priority to more urgent tasks, to maximize hit ratio. Whereas using the inverted cost (similar to SRPT) gives higher priority to tasks that would minimize response time.

B. Scheduler Priority Queue

Each task that has no dependency, i.e., depends on no update-tasks, is inserted in the priority queue to represent a workflow. Upon task addition, (1) task dependency information is detected, and (2) if existing task priority changes (e.g., if the new task is a query then the deadline of update tasks it depends upon may get updated), then (a) the task is inserted in the list of tasks, and (b) the priority queue is updated.

The priority queue is updated when a new ready task is inserted, and when existing tasks' priority change, to reflect new priorities. At each scheduling point, the task on top of the priority queue is selected as the next task to be executed in secondary database. Note that for parallel tasks we dispatch a number of tasks that equals to the execution parallel degree.

Upon completion of a task, the workflow is updated: the ready task is deleted from list of tasks and from the priority queue. The dependency information is updated for the task(s) that depends on this completed task. If there are new ready task(s), then they are inserted into the priority queue.

Whenever the priority queue is updated, all the query-tasks in the priority queue below where the update took place are examined if they can still meet the deadline. If not, the task is scheduled to fallback, as it would take longer if it waits to get executed in the secondary database.

C. Cost Model

A precise estimation of query and DML costs is crucial for the success of our proposed scheduler, which makes cost-based decisions. Estimating a query cost is a very hard problem. However, similar to query optimizers,

 ISBN: 978-1-61208-637-8

all that we need is a reliable cost model that enables us to compare costs of different tasks. That is, accuracy of the cost-model is relative in a sense. Therefore, we propose to use the compiler's estimated cost as the cost of a query task. In particular, the cost of a query fragment if executed in the secondary database is the query-task cost. Similarly, we use the primary database cost of this query fragment (fallback cost) to be the deadline of this query-task.

Estimating the fallback cost is not straightforward because we expose both the access path and the join order in the secondary system during query optimization in the primary. That is, there is no separate in-primary vs. in-secondary query optimization. As a result, we can retain an access path type - such as full table scan vs. index-based scan - as well as the join order that were estimated as best in the secondary system, while later on during optimization we decide to execute a larger fragment in the primary system. To provide accurate fallback cost the optimization phase had to be enhanced to remember at any point the best purely in-primary cost of each access path and join.

For update-tasks, factors that affect the cost include: (1) update granularity: cardinality of the delta relation, (2) network bandwidth/speed, (3) overhead to prepare the changes, and (4) overhead to apply the changes at the secondary database.

These cost factors are extremely hard to estimate and are workload and system-specific. Our proposed effective and accurate cost model is simply the moving-average load rate. Specifically, we maintain the load-rate as follows. While loading the table for first time to the secondary database, we observe the overall load rate *LR*, which measures how long it takes to load a single row of that table, on average. Then, we use *LR* times the number of updated rows as the cost for each DML task on that table. Once this change is uploaded, we measure the actual new load rate: *LR'*. *LR* is then updated to be the average of *LR* and *LR'* to have a new, more accurate load rate to estimate the cost of future DML tasks on this table. This way, if a table grows over time that its updates become more expensive, this will be captured by this cost model, and vice versa. In the next section we detail why it is possible for us to know the accurate number of rows for each DML task and why we do not need to rely on any typical secondary structure – like indexes – for this purpose.

V. IMPLEMENTED PROTOTYPE

We have implemented a prototype [18] of the proposed scheduler infrastructure and the above detailed scheduling policy in the Oracle general purpose database system as the primary database and the recent RAPID Data Processing Unit (DPU)-based accelerator system developed at Oracle Labs [19]
[20]. The RAPID accelerator is a main-memory system with a bandwidth-optimized architecture for big data computation. Relations in the primary are loaded into the DPUs at a given SCN, by reformatting the data in a hybrid-columnar format. In the primary database, changes post initial load are represented in memory resident transactional journals, just as typically maintained for in-memory optimized RDBMS relations [9]. As rows are logged into the journals, corresponding tasks are defined and messaged to the scheduler.

The scheduler is designed to run on both the primary and the secondary (the accelerator). The primary-side scheduler is the main scheduler that captures and maintains dependency and priority information, and decides when and what queries and DMLs to push to RAPID. It also communicates the dependency and priority information to the RAPID-side scheduler. During the initial load into the secondary system, data is scanned in parallel from the primary instance; whether the scan happens through the buffer cache, or from direct path from secondary storage, or directly from in-memory compression units using the IMCU Oracle format, we read the data at the level of the scan row source level; at this point data is vectorized, encodings can be applied, and distributed to the secondary system. In this process we know the exact number of rows we load into the Rapid nodes, and we maintain this information, together with encoding statistics, into a segment of each instance shared memory, which we call the *Rapid global state*. For each table loaded into the secondary system we enable in-memory journal tracking – by using the already developed Oracle mechanism to keep in the shared memory of each Oracle instance per relation journals of changed rows at their corresponding SCNs. Note that this journaling activity happens even if we do not require the relation to be maintained in IMCU formats – that is, for the purpose of the prototyped in-memory journaling feature has been decoupled from in-memory data encoding. As the journals are scanned and DML tasks are generated, we keep track of the exact number of rows that have been journaled at each SCN, per relation. This information is available therefore to the scheduler, and it is also used to update the number of rows loaded into the secondary and compression statistics in the Rapid global state. Once the global state statistics are updated and changes are acknowledged applied by the Rapid nodes, the in-memory journals can be truncated in case of memory pressure. For the case of direct insert/load, we make an exception and do not journal the changes; instead, entire data blocks for the appended data are scanned and changes are applied in Rapid before the transaction ends in the primary; nevertheless, in this case we also know the exact number of rows sent to Rapid and we can update the statistics in the global state. On a general note, there is no concept of pieced row in Rapid, as data is maintained in hybrid columnar format. If a pieced row is encountered during initial load or during direct path insert, the next piece is scanned recursively until the entire row is constructed, and then each column within the row is added to the respective column vector, before the vector is compressed. When scanning journaled rows, if a row is pieced we

 ISBN: 978-1-61208-637-8

similarly traverse the links in the journal for retrieving all the pieces. The typical case when pieced rows are recurrent is when the shape of the relation is such that majority of rows are pieced (for instance, when the number of columns exceeds a certain limit); in such cases, the initial load as well as DML tasks significantly involve pieced rows. As the per-row initial cost for a relation is computed during initial load, for such typical cases this cost reflects the overhead of handling multiple pieces before forming the full row image and this overhead applies to DML tasks as well.

The RAPID-side scheduler is a distributed independent version of the scheduler that acts as an actuator: independently on each DPU, a scheduler instance makes sure that tasks are run in the right order communicated by the primary-side scheduler. By running independently, we avoid any overhead of coordination or hand-shaking protocols.

The primary-side scheduler was implemented as a background process. We used basic data structures to implement the scheduler metadata. In particular, the list of tasks and priority queue were implemented as linked lists. We relied on existing database system layer for inter-process communication. The goal of this prototype was to prove functionality and the relative benefits of the scheduler module. Our implementation was for single-instance and no parallelism, for the primary-side scheduler. In this prototype, we implemented the cost model and scheduling policy explained above. However, we implemented the hooks to enable the addition of different scheduling policies. In particular, the scheduling policy is configured as an Oracle startup parameter.

D. Preliminary Results

Using the 22 standard TPC-H [28] queries and the TPC-H refresh streams, we were able to demonstrate that the scheduler sometimes decides for certain query-tasks to fallback to primary database, when it is pending on many update tasks that cost more than the query deadline. Surprisingly this was the case for very small or simple queries, in addition to the intuitive case where query is pending on large updates. The reason is that for simple queries, the cost is typically very small, and hence the deadline is very close. Thus, for any reasonably large update, the query would fall back to the primary system.

We also measured the scheduler overhead on X4 machines, and with this basic implementation and experiments on small scale TPC-H (up to F=64), we found that the total query overhead ranges between 50 and 100 micro-seconds. Most of this overhead is due to inter-process communication, whereas the overhead of maintaining the scheduler metadata (i.e., the priority queue and the list of tasks) was in the order of few micro-seconds. Further, the overhead of maintaining the list of task as the priority queue grows was almost flat, i.e., grows very slowly as the priority queue grows. This shows that a scheduler module is feasible, beneficial, and has very minimal overhead that can be further optimized. The expectation is that the scheduler overhead and load do not correlate to the workload size, since an update-task or a query-task is still a single task weather it processes gigabyte or petabytes of data.

VI. CONCLUSIONS

In this paper, we proposed a novel infrastructure for prioritizing and scheduling queries and updates between a primary and secondary database. We detailed the system model and architecture for single instance primary database case. We gave details of our implemented prototype in which we adopt a scheduling policy that maximizes hit ratio and minimizes response time. The runtime overhead incurred due to the scheduler activity was measured using TPC- H queries, with no code optimizations, and results show that the scheduler module has negligible overhead. This demonstrates that the scheduler is feasible, beneficial and effective.

ACKNOWLEDGEMENTS

We would like to acknowledge Kantikiran Pasupuleti, Seema Sundara, Adrian Schuepbach, and Benjamin Schlegel for insightful discussions and feedback.

1 Work was done while authors were member of Oracle Labs, Oracle, USA

REFERENCES

1. Oracle Patent Disclosure: A. Di Blas, B. Schlegel, S. Idicula, S. Petride and K. Pasupuleti, N. Agarwal Method for Consistent Concurrent Execution of Multiple Queries in Presence of Base Table Updates, Disclosed Jan 2015.
2. G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In Proc. of RTSS ’95, pp. 90-99, 1995.
3. S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In ICDE, pp. 357–368, 2009.
4. W. Cao and D. Aksoy. Beat the clock: a multiple attribute approach for scheduling data broadcast. In MobiDE ’05, pp. 89–96, 2005.
5. D.-Z. He, F.-Y. Wang, W. Li and X.-W. Zhang. Hybrid earliest deadline first preemption threshold scheduling for real-time systems. In *ICMLC*, pp. 433-438, 2004.
6. B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. ACM Trans. Inter. Tech., 6(1), pp. 20–52, 2006.
7. T. Lahiri, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase et. Al., Oracle Database In-Memory: A dual format in-memory database, ICDE 2015: 1253-1258
8. G. Marchionini, Exploratory search: from finding to understanding. Commun. ACM 49(4), pp. 41-46, 2006.
9. J. Erickson, “In-Memory Acceleration for the Real-Time Enterprise”,

http://www.oracle.com/us/corporate/features/database-in-memory-option/index.html [retrieved: 04, 2018].

10. W. Vogels, "Eventually Consistent", CACM, pp. 40-44, 2009.
11. A Labrinidis and N Roussopoulos, "Balancing performance and data freshness in web database servers", VLDB, pp. 393-404, 2003.
12. Response Time references(s)
13. Oracle® Flashback Technologies, http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html [retrieved: 04, 2018].
14. Oracle® TimesTen In-Memory Database and TimesTen Application-Tier Database Cache, http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/index.html [retrieved: 04, 2018].
15. Oracle® TimesTen Application-Tier Database Cache User's Guide: http://st-doc.us.oracle.com/12/12102/TTCAC/define.htm#TTCAC211 [retrieved: 04, 2018].
16. V. Sikka, F. Farber, W. Lehner, S. K. Cha, T. Peh and C. Bornhovd, "Efficient transaction processing in SAP HANA database: the end of a column store myth", SIGMOD, pp. 731-742, 2012.
17. H. Ma, W. Qian, F. Xia, J. Wei, C. Yu and A. Zhou, On benchmarking online social media analytical queries. In First International Workshop on Graph Data Management Experiences and Systems (GRADES), ACM, Article 10 , pp. 1-7, 2013.
18. S. Guirguis and S. Petride, Query and Change-Propagation Scheduling: A Real-Time Scheduling Model for Heterogeneous Database Systems, Patent Application, 2017.
19. V. Govindaraju, S. Idicula, S. Agrawal, V. Vardarajan, A. Raghavan, J. Wen et. al,, Big Data Processing: Scalability with Extreme Single-Node Performance, IEEE Big Data Congress, pp. 129-136, 2017.
20. S. R Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, et. al., A many-core architecture for in-memory data processing, IEEE/ACM MICRO, pp. 245-258, 2017.
21. V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, et. al., DB2 with BLU Acceleration: So Much More than Just a Column Store, VLDB Volume 6 Issue 11, pp. 1080-1091, 2014.
22. D. J. Abadi, P. A. Boncz and S. Harizopoulos. 2009. Column-oriented Database Systems. Proc. VLDB Endow. 2, 2 (Aug. 2009), pp. 1664–1665. https://doi.org/10.14778/1687553.1687625.
23. K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan and T. F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). ACM, New York, NY, USA, pp. 36–47. https://doi.org/10.1145/2485922.2485926 [retrieved: 04, 2018].
24. C. Root and T. Mostak. 2016. MapD: a GPU-powered big data analytics and visualization platform. In ACM SIGGRAPH 2016 Talks (SIGGRAPH '16). ACM, New York, NY, USA, Article 73, pp. 1-2, DOI: https://doi.org/10.1145/2897839.2927468 [retrieved: 04, 2018].
25. C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs and T. C. Mowry, A scalability service for dynamic web applications. CIDR, 2005.
26. Y. Huang, R. Sloan and O. Wolfson. Divergence Caching in Client Server Architectures. In Proc. 3rd International Conference on Parallel and Distributed Information Systems, 1994.
27. T. Malik, X. Wang, P. Little, A. Chaudhary and A. Thakar, A Dynamic Data Middleware Cache for Rapidly-growing Scientific Repositories, arXiv.org, arXiv:1009.3665
28. TPC-H benchmark, www.tpc.org/tpch [retrieved: 04, 2018

 ISBN: 978-1-61208-637-8

 ISBN: 978-1-61208-637-8