Adaptive Concurrent Query Execution Framework for an Analytical In-Memory Database System - Supplemental Material

Harshad Deshmukh Hakan Memisoglu Jignesh M. Patel
University of Wisconsin - Madison
{harshad, memisoglu, jignesh}@cs.wisc.edu

***Abstract*—This document supplements the paper [1]. The supplemental material includes detailed explanations of certain concepts, some clarifying examples, the DAG traversal algorithm used by the scheduler and finally some additional experiments.**

I. Work Orders

Work done for executing a query in Quickstep is split into multiple *work orders*. A work order contains all the information that is needed to process tuples in a given data block. A work order encapsulates the relational operator that is being applied, the relevant input relation(s), location of the input data block, any predicate(s) to be applied on the tuples in the input block, and descriptors to other run-time structures (such as hash tables).

Consider the following full table scan query to illustrate the work order concept:

```
SELECT name FROM Employee WHERE city='San Diego'
```

The plan for this query has a simple *selection* operator. For the selection operator, the number of work orders is same as the number of input blocks in the `Employee` table. Each selection work order contains the following information:

- Relation: `Employee`, attribute: `name`
- Predicate: `city='San Diego'`
- The unique ID of an input block from the `Employee` table

The work orders for a join operation are slightly more complicated. For example, a *probe work order*, contains the unique ID of the probe block, a pointer to the hash table, the projected attributes, and the join predicate. Each operator algorithm (e.g. a scan or the build/probe phase of a hash-join) in the system has a C++ class that is derived from a root abstract base class that has a virtual method called `execute()`. Executing a work order simply involves calling the `execute()` method on the appropriate operator algorithm C++ class object.

II. Storage Management in Quickstep

Data organization in the Quickstep storage manager holds the key to intra-query parallelism [2]. Data in a relation are organized in the form of blocks. Each block holds a collection of tuples from a single table. A unique aspect of the storage organization in Quickstep is that blocks are considered to be independent and self-contained mini-databases. Thus, when creating an index, instead of creating a global index with "pointers" to the tuples in blocks, the index fragments are stored within the blocks. Each block is internally organized into *sub-blocks*. There is one *tuple storage sub-block*, which could be in a row store or a column store format. In addition, each block has one sub-block for each index created on the table. CSB+-tree [3] and BitWeaving [4] indices are currently supported. The blocks are free to self-organize themselves and thus a given table may have blocks in different formats. For example, new blocks in a table may be in a row store format, while older blocks may be in a column store format.

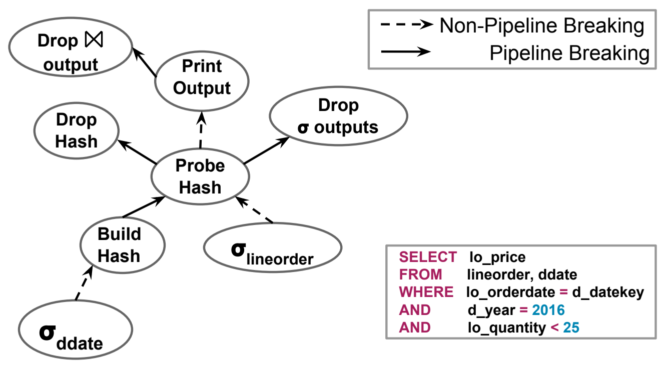
This storage block design, as articulated earlier in [2] enables the query execution to be broken down into a set of independent tasks on each block. This is a crucial aspect that we leverage in the design of our scheduler.

The storage manager also contains a *buffer pool manager*. It organizes the memory as an array of slots, and overlays blocks on top of the slots (so block sizes are constrained to be a multiple of the underlying slot size). Memory allocations for data blocks for *both* permanent and temporary tables are always made from a centralized buffer pool. In addition, all allocations for run-time data structures, such as hash tables are also made by the buffer pool. The buffer pool manager employs an LRU-2 replacement policy. Thus, it is possible for a hash table to get evicted to disk, if it has become "cold"; e.g. if it belongs to a suspended query.

III. DAG Traversal Algorithm

The Query Manager is presented with a DAG for each query, where each node in the DAG represents a relational operator primitive. The edges in the DAG are annotated with whether the *consumer* operator is blocked on the output produced by the *producer* operator, or whether data pipelining is allowed between two adjacent operators.

Consider a sample join query and its DAG showed in Figure 1. The solid arrows in the DAG correspond to "blocking" dependencies, and the dashed arrows indicate pipeline-



```
SELECT  lo_price
FROM    lineorder, ddate
WHERE   lo_orderdate = d_datekey
AND     d_year = 2016
AND     lo_quantity < 25
```

Fig. 1. A join query and its DAG

able/non-blocking dependencies. To execute this query we need to select tuples from the `ddate` table, stream them to a hash table, which can then be probed by tuples that are created by the selection operator on the `lineorder` table. The output of the probe hash operation can be sent to the print operator, which displays the result. Note that the "drop hash" operator is used to drop the hash table, but only after the "probe hash" operation is complete. Similarly, the other drop operators indicate when intermediate data can be deleted.

Algorithm 1 DAG Traversal

```
1:  G = {V, E}
2:  activeEdges = {e ∈ E | e.isNotPipelineBreaking()}
3:  inactiveEdges = {e ∈ E | e.isPipelineBreaking()}
4:  completedNodes = {}
5:  for v ∈ V do:
6:      if v.allIncomingEdgesActive() then
7:          v.active = True
8:      else
9:          v.active = False
10: while completedNodes.size() < V.size() do
11:     for v ∈ V − completedNodes do
12:         if v.allIncomingEdgesActive() then
13:             v.active = True
14:         if v.active then
15:             v.getAllNewWorkOrders()
16:             if v.finishedGeneratingWorkOrders() then
17:                 completedNodes.add(v)
18:                 for outEdge ∈ v.outgoingEdges() do
19:                     activeEdges.add(outEdge)
```

The Query Manager uses a DAG Traversal algorithm (cf. Algorithm 1) to process the DAG, which essentially is an iterative graph traversal method. The algorithm simply finds nodes in the DAG that have all their dependencies met, and marks such nodes as "active" (line 13). Work orders are requested and scheduled for all active nodes (line 15), and the completion of work orders is monitored. Operators are stateful and they produce work orders when they have the necessary data. The work order generation stops (line 16) when the operators no longer have any input to produce additional work orders. When no more work orders can be generated for a node, that node is marked as "completed" (line 17). When a node is marked as completed, all outgoing blocking edges (the solid lines in Figure 1) are "activated" (line 19). Pipelining is achieved as all non-blocking edges (dotted lines in Figure 1) are marked as active upfront (line 2). The query is deemed as completed when all nodes are marked as "completed."

IV. Resource Map Discussion

An example Resource Map of an incoming query to the system is shown below:

```
CPU:    {min: 1 Core, max: 20 Cores}
Memory: {min: 20 MB,  max: 100 MB}
```

This Resource Map states that the query can use 1 to 20 cores (i.e. specifies the range of intra-operator parallelism) and is estimated to require a minimum of 20 MB of memory to run, and an estimated 100 MB of memory in the worst case.

In Quickstep, the query optimizer provides the estimated memory requirements for a given query. Other methods can also be used, such as inferring the estimated resources from the previous runs of the query or other statistical analyses. The scheduler is agnostic to how these estimates are calculated.

V. Pipelining in Quickstep

During a work order (presumably belonging to a producer relational operator in a pipeline) execution, output data may be created (in blocks in the buffer pool). When an output data block is filled, the worker thread sends a "block filled" message to the corresponding query's manager via the following channel: Worker → Foreman → Policy Enforcer → Query Manager. The Query Manager may then create a new work order (for a consumer relational operator in the pipeline) based on this information; e.g. if this block should be pipelined to another operator in the query plan.

Note that pipelining in Quickstep works on a block-basis, instead of the traditional tuple-basis.

VI. Thread Model

Quickstep currently runs as a single server process, with multiple user-space threads. There are two kinds of threads. There is one *Scheduler* thread, and a pool of *Worker* threads. All the components in the scheduler architecture except the worker thread pool run in the scheduler thread. In the current implementation, all threads are spun upfront when the database server process is instantiated, and stay alive until the server process terminates.

The threads use the same address space and use shared-memory semantics for data access. In fact the buffer pool is stored in shared memory, which is accessible by all the threads. Each worker thread is typically pinned to a CPU core. Such pinning avoids costs incurred when a thread migrates from one CPU core to another, which results in loss of data and instruction cache locality. We do not pin the scheduler thread, as its CPU utilization is low and it is not worth dedicating a CPU core for the scheduler thread.

Every worker thread receives a work order from the Foreman, executes it and then waits for the next work order. In order to minimize worker's idle time, typically each worker is issued multiple work orders at any given time. Thread-safe queues are used to communicate between the threads. The communication happens through light-weight messages from the sender to the receiver thread, which is internally implemented as placing a message object on the receiver's queue. A receiver reads messages from its queue. A thread (and its queue) is uniquely identified by its thread ID.

The thread communication infrastructure also implements additional features like the ability to query the lengths of any queue in the system, and cancellation of an unread message. For simplicity, we omit discussion of these aspects.

VII. Motivation for the Learning Agent Module

One might question the need of the Learning Agent and instead consider assigning a fixed probability value to each query (say $1/N$, with N queries in the fair policy). In the following section, we address this issue. A motivational example for the learning agent is described in Appendix VII.

We perform an experiment, where the goal is to analyze the patterns in work order execution times of two queries. The dataset used for the experiment comes from the Star Schema Benchmark (SSB) [5] at a scale factor of 100. We pick two SSB queries $Q1.1$ and $Q4.1$, and execute them on a machine with 40 CPU cores. $Q1.1$ has a single join operation and $Q4.1$ has four join operations. Figure 2 shows the observed average time per work order for both queries. We now describe the trends in time per work order for the queries.

We can observe $Q1.1$'s execution pattern denoted by the dashed line in Figure 2. The time per work order remains fairly stable (barring some intermittent fluctuations) from the beginning until 1.8 s. This phase corresponds to the selection operation in $Q1.1$ which evaluates predicates on the *lineorder* (fact) table. A small bump in time per work order can be observed at the 1.8 s mark, when the probe phase of $Q1.1$ begins and continues until 2 s. Towards the end of the execution of $Q1.1$, (2.2 s) there is a spike in time per work order when the query enters the aggregation phase. The output of the hash join is fed to the aggregation operation. The results of aggregation are stored in per-thread private hash tables, which are later merged to produce the final output.

Now we analyze the execution pattern for $Q4.1$ which has 4 join operations. This query is more complex than $Q1.1$.Therefore, the execution pattern of $Q4.1$ exhibits more phases, with different times per work order as compared to $Q1.1$. Various small phases before the 0.5 s mark correspond to the selection predicates that are applied to the dimension tables. (Note that in $Q4.1$ there is no selection filter on

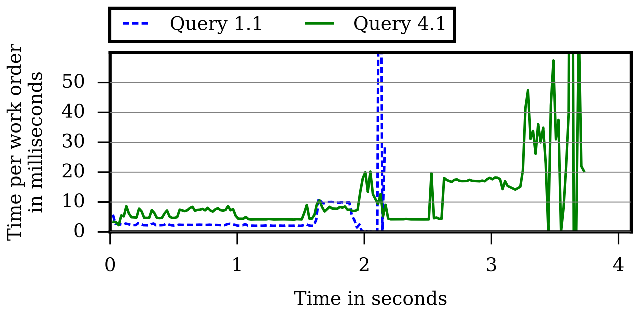


Fig. 2. Time per work order for Q1.1 and Q4.1

the *lineorder* table). The selections on dimension tables get executed quickly. The longer phases denote the different probe hash table operations in the query. Towards the end, similar to $Q1.1$, there is a spike in the execution time per work order which correspond to the aggregation phase.

It is clear that the work order execution times for both queries are different, and the difference between them changes over time. If the scheduler assigns the same probability to both queries (i.e. 0.5), it is equally likely to schedule a work order from either of them. As a result, the queries will have different CPU utilization times in a given epoch, thus resulting in an unfair CPU allocation. In order to be consistently fair in allocating CPU resources to the queries, we should continuously observe the work order execution times of queries and adjust the CPU allocation accordingly.

VIII. Usage of Linear Regression in Learning Agent

The Learning Agent uses linear regression for predicting the execution time of the future work orders. To lower the CPU and memory overhead of the model, we limit the amount of execution statistics stored in the Learning Agent. We discard records beyond a certain time window. When all the work orders of an operator finish execution, we remove its records completely. In a query, if multiple relational operators are active, linear regression combines the statistics of all active operators and predicts a single value for the next work order execution time.

IX. Resource Choices for Policy Implementations and Load Controller Implementations

In the current implementation of Quickstep, we have focused on two key types of resource in the in-memory deployment scenarios – CPU and memory. Both these resources have different resource characteristics, which we outline below.

First, consider the CPU resource. On modern commodity servers there are often tens of CPU cores per socket, and the aggregate number of cycles available per unit time (e.g. a millisecond) across all the cores is very large. Further, an implication of Quickstep's fine-grained task allocation and execution paradigm is that the CPU resource can be easily shared at a fine time-granularity. Several work orders, each from different query can be executed concurrently on different CPU cores, and each query may execute thousands or millions or even more number of work orders. Thus, in practical terms, the CPU resource can be viewed as a nearly infinitely divisible resource across concurrent queries. In addition, overall system utilization is often measured in terms of the CPU utilization. Combining all these factors, specifying a policy in terms of the CPU utilization is natural, and intuitive for a user to understand the policy. For example, saying that a fair policy equally distributes the CPU resource across all (admitted) concurrent queries is simple to understand and reason.

Memory, on the other hand, is a resource that is allocated by queries in larger granular chunks. Active queries can have varying memory footprints (and the footprint for a query can change over the course of its execution). Thus, memory as a resource is more naturally viewed as a "gating" resource. Therefore, it is natural to use it in the load controller to determine if a query can be admitted based on its requested memory size. Actual memory consumption for queries can also be easily monitored, and when needed queries can be suspended if memory resource needs to be freed up (for some other query, perhaps with a higher priority).

X. Applicability of SSB for our evaluation

The SSB is based on the TPC-H benchmark, and is designed to measure the query performance when the data warehouse uses the popular Kimball [6] approach. At a scale factor of X, the benchmark corresponds to about X GB of data in the corresponding TPC-H warehouse. The SSB benchmark has 13 queries, divided in four categories. Each query is identified as **qX.Y**, where X is the class and Y is the query number within the class. There are four query classes, i.e. $1 \leq X \leq 4$. The first and second classes have three queries each, the third class has four queries, and the fourth class has three queries. The queries in each category are similar with respect to aspects such as the number of joins in the query, the relations being joined, the filter and aggregation attributes. The grouping of queries in various classes makes this benchmark suitable for our experiments, as it provides a way of assigning priorities to the queries based on their class.

XI. Evaluation of Proportional Priority Policy

Now we examine the scheduler's behavior to the proportional priority policy in which the higher priority integer implies higher importance.

We pick two queries from each SSB class, and assign them a priority value. Our priority assignment reflects the complexity of the queries from the corresponding class. For instance, query class 1 has one join, class 2 has two joins and so on. Recall that in our implementation a higher priority integer implies higher importance.

Figure 3 shows the CPU allocation among concurrent queries in the proportional priority policy. We can see that a

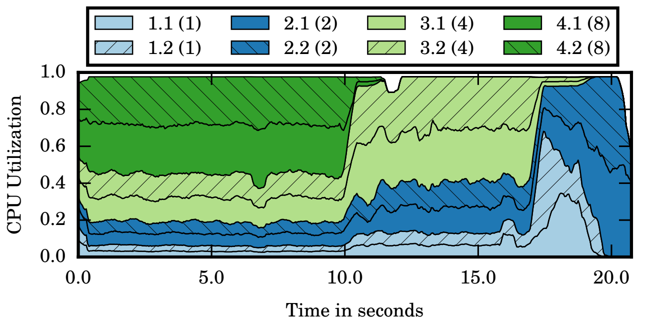
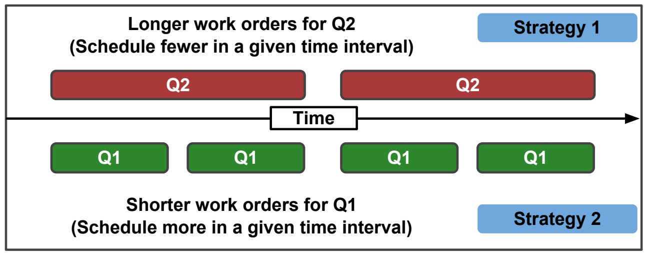


Fig. 3. CPU allocation for proportional priority policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority N



Longer work orders for Q2
(Schedule fewer in a given time interval)
Strategy 1
Q2
Q2
Time
Q1
Q1
Q1
Q1
Shorter work orders for Q1
(Schedule more in a given time interval)
Strategy 2

Fig. 4. Scheduling queries having different work order execution times for the fair policy. The solid boxes with Q_i depicts the lifetime of a work order from the Query Q_i.

higher priority query gets proportionally higher share of CPU as compared to the lower priority queries. When all queries from the priority class 8 finish their execution (11 seconds), the lower priority classes elastically increase their CPU utilization, so as to use all the CPU resources. Also note that among the queries belonging to the same class, the CPU utilization is nearly the same, as described in the policy specifications.

XII. Policy Enforcer

The Policy Enforcer applies a high level policy for resource allocation among concurrent queries. It uses a probabilistic-framework to select work orders from a pool of work orders belonging to different concurrent queries for scheduling. The Policy Enforcer assigns a probability to each active query, which indicates the likelihood of a work order from that query getting scheduled for execution in the near future. The probability-based work order selection strategy brings powerful control to the scheduler through a single parameter – i.e. by controlling the probability setting, the scheduler can control the resource sharing among concurrent queries.

The challenge in designing the policy enforcer lies in transforming the policy specifications to a set of probabilities. A critical piece that we use in such transformations is the prediction of work order execution times for the concurrent queries, which is done by the Learning Agent. In the remainder of this section, we provide an intuition for deriving probability values from the work order execution times.

We now motivate the probabilistic approach used by the Policy Enforcer with an example. Consider a single CPU core and two concurrent queries q_1 and q_2. (The idea can be extended to multi-cores and more than two queries.) Initially, we assume perfect knowledge of the execution times of work orders of the queries. Later, we will relax this assumption.

Let us assume that as per the policy specifications, in a given time interval, the CPU resources should be shared equally. Suppose that work orders for q_1 take less time to execute than work orders for q_2, as shown in Figure 4. As the Policy Enforcer aims to allocate equal share of the CPU to q_1 and q_2, a simple strategy can be to schedule proportionally more work orders of q_1 than those of q_2, in a given time window. The number of scheduled work orders is inversely related to the work order execution time. This proportion can be determined by the probabilities pb_1 and pb_2 for queries q_1 and q_2, respectively. The probability pb_i is the likelihood of the scheduler scheduling next work order from query i. The probability is assigned by the Policy Enforcer to each active query in the system. Note that, $pb_1 > pb_2$ and $pb_1 + pb_2 = 1$.

Notice that the Policy Enforcer is not concerned with the complexities of the operators in the query DAGs. It simply maintains the probability associated with each active query which is determined by the query's predicted work order execution times.

The Policy Enforcer can also function with workloads that consist of queries categorized in multiple classes, where each class has a different level of "importance" or "priority". The policy specifies that the resource allocation to a query class must simply be in accordance to its importance, i.e. queries in a more important class should collectively get a higher share of the resources, and vice versa. In such scenarios, the Policy Enforcer splits its work order selection strategy in two steps - selection of a query class and subsequent selection of a query within the chosen query class. Intuitively, the Policy Enforcer should assign higher probability to the more important class and lower probability to the less important class.

Once a query class is chosen, the Policy Enforcer must pick a query from the chosen class. Each query class can specify an optional intra-class resource allocation sub-policy. By default, all queries within a class are treated equally. Thus, the probability-based paradigm can be used to control both inter and intra-class resource allocations.

There could be many reasons for categorizing queries in classes, including the need to associate some form of urgency (e.g. interactive vs batch queries), or marking the importance of the query source (e.g. the position of the query submitter in an organizational hierarchy). In addition, the resource allocations across different classes can also be chosen based on various scales, such as linear or exponential scale allocations based on the class number. An attractive feature of the Policy Enforcer is that it can be easily configured for use in a variety of ways. Under the covers, the Policy Enforcer simply maps each class to a collective class probability value, and then maps each query in each class to another probability. Once these probabilities are calculated, the remaining mechanisms simply use them to appropriately allocate resources to achieve the desired policy goal.

REFERENCES

[1] H. Deshmukh, H. Memisoglu *et al.*, "Adaptive concurrent query execution framework for an analytical in-memory database system."

[2] C. Chasseur and J. M. Patel, "Design and evaluation of storage organizations for read-optimized main memory databases," *PVLDB*, 2013.

[3] J. Rao and K. A. Ross, "Making B$^+$-trees cache conscious in main memory," in *SIGMOD*, 2000.

[4] Y. Li and J. M. Patel, "Bitweaving: fast scans for main memory data processing," in *SIGMOD*, 2013.

[5] P. O'Neil, E. O'Neil *et al.*, "The star schema benchmark," http://www.cs.umb.edu/~poneil/StarSchemaB.pdf, Jan 2007.

[6] R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*, 2nd ed., 2002.