Comet: Batched Stream Processing in Data Intensive Distributed Computing

Bingsheng He[†] Mao Yang[†] Zhenyu Guo[†] Rishan Chen[‡]
Bing Su[†] Wei Lin[†] Lidong Zhou[†]
[†]*Microsoft Research Asia* [‡]*Beijing University*

ABSTRACT

Performance and resource optimization is an important research problem in data intensive distributed computing. We present a new *batched stream processing* model that captures query correlations to expose I/O and computation redundancies for optimizations. The model is inspired by our empirical study on a trace from a production large-scale data processing cluster, which reveals significant redundancies caused by strong temporal and spatial correlations among queries.

We have developed Comet, a query processing system that embraces the batched stream processing model for optimizations. We have integrated Comet with DryadLINQ. With its roots in query optimizations for database systems, Comet enables a set of new heuristics and opportunities tailored for distributed computing in DryadLINQ. Optimizations in Comet are effective. The evaluation of a micro-benchmark on a 40-machine cluster shows a 42% reduction in total machine time and over 40% reduction in total I/O. Our simulation on a real trace covering over 19 million machine hours shows an estimated I/O saving of over 50%.

1 INTRODUCTION

Data intensive scalable computing (DISC) systems, such as MapReduce/Sawzall [8, 20], Dryad/DryadLINQ [16, 28], Hadoop/Pig [14, 19] and SCOPE [6], have unanimously embraced the *batch* processing model, where each *query* specifies computation on a large bulk of data. These systems tend to consider queries individually. In reality, we are facing the challenging system problem of executing a large number of complicated queries on a large amount of data every day across thousands of servers. Performance and resource optimization of all those queries is essential for effective utilization of business investment of millions of dollars. In this paper, we study the performance and resource optimization in a DISC system.

We have examined a 3-month trace from a production cluster dedicated to large-scale data processing. This trace captures a workload consisting of 13 thousands queries whose executions took a total of 19 million machine hours. Our study of the trace shows that the efficiency of the system is far from ideal. For example, we find that over 50% of the total I/O are caused by repetitive scans on input data, and by repetitive computation among different queries. Significant I/O redundancies cause significant waste in the disk bandwidth, and also significant waste in total machine time.

Further study of the trace uncovers that the redundancy is due to correlations among queries. The workload exhibits *temporal correlations*, where it is common to have a series of queries involving the same recurring computation on the same data stream in different time windows. The workload further exhibits *spatial correlations*, where a data stream is often the target of multiple queries involving different but somewhat overlapping computation. The detailed statistics show that popularity of data streams and their fields follows the *power-law* distributions closely; similar distributions also show up in the custom functions that are used in the queries. As a result, the queries are heavily concentrated on a relatively small number of data streams and fields, and tend to use the same set of custom functions.

Correlations among queries in DISC system naturally lead us to look for solutions and inspirations from existing cross-query or cross-job optimizations in databases systems and (distributed) operating systems. Olston et al. [18] and Popa [21] have recently proposed to apply the existing techniques in those fields for optimizations. For example, query result *caching* or perfect *view matching* [2] is particularly effective in identifying common computations across queries and in allowing reuse of computed results.

While leveraging proven concepts is clearly a step in the right direction, applying them in the current computing environment itself is challenging due to the inherent complexity and unpredictability in such systems [18]. For example, current query optimizations in DISC systems are usually rule- or heuristic-based, in contrast that a query is often complicated and contains user-defined functions (i.e., custom functions) with little or even unknown cost characteristics; queries are executed when received, with little consideration given to previous or future executions. All those factors limit the effectiveness of traditional optimization mechanisms. For example, our simulation shows that query result *caching* or perfect *view matching* [2] removes only 15% of the I/O redundancy in the trace.

To unleash optimization opportunities of traditional mechanisms, we introduce the *Batched Stream Processing* model (or BSP in short) to characterize recurring computation on incrementally appended data streams. The recurring computation on the same data stream

forms a *query series*. A query series captures a sequence of the same computation on different sets of segments of the same stream: queries in the same query series are by definition correlated. Query series makes the occurrence of these queries predictable. Due to the power-law data and computation popularity, queries in different query series have a high probability in sharing the same I/O to scan the input data and in sharing common computation. Those queries can be scheduled to run together as a single combined query, thereby removing redundancies.

We have built *Comet*, a system to allow users to submit query series and to enable a set of new global optimizations for DISC applications. In Comet, computation is triggered by arrivals of new bulk updates to the streams. This is in contrast to current systems, where queries are executed upon their arrivals. Comet aligns the computation units from different query series into a single *jumbo query*, to maximize sharing. By exploiting the sharing within jumbo queries, Comet brings new relevance to the traditional database optimization techniques, especially those for continuous queries [26] and multiple queries [25], to DISC systems. Comet enriches these traditional optimization techniques with its special attention to the distributed nature of the DISC systems: for example, Comet's physical optimizations use co-location and local data reduction for reducing the network traffic.

We have integrated Comet into the logical and physical optimizations in DryadLINQ [28]. We have modified DryadLINQ implementation to enable Comet-specific optimizations, including normalization and optimizations on a jumbo query. The flexibility in the LINQ language is a frequent source of headaches for implementation: imperative custom functions in DryadLINQ pollutes the otherwise declarative representation of queries. As a result, some optimizations that are natural in database systems need special care to achieve.

We used two complementary methods to evaluate the effectiveness of Comet. Our empirical studies with a micro benchmark show a reduction of over 40% with the optimizations in Comet. Our simulation results show a reduction of over 50% with the optimizations in Comet.

The contributions of this paper are as follows: first, we report interesting new findings from a real production trace, which reveals performance and resource problems in current DISC systems. Second, we propose the execution model for batched stream processing to unleash the power of optimizations, addressing the existing performance and resource problems. Finally, we implement a prototype and a simulator to validate the feasibility and effectiveness of our model and optimizations.

The rest of the paper is organized as follows. Section 2 describes our empirical study on a real-world workload from a deployed DISC cluster. We present the Comet execution model in Section 3. Sections 4 documents the details of integrating Comet into DryadLINQ. Section 5 presents the implementation details of the simulator, followed by the experimental results in Section 6. We then review the related work in Section 7, with discussions in Section 8. Finally, we conclude in Section 9.

2 An Empirical Study

We have obtained a 3-month query trace from a deployed DISC system with over thousands of machines. The queries are mainly data mining tasks on logs generated from a wide range of services within the company, such as search query/click logs. The trace documents the information related to executions of all query jobs in the system. The information includes query itself, submission time, query plan, and performance statistics, such as the amount of I/O of each step in the query plan. The trace contains nearly 13,000 of successfully executed queries, taking a total of 19 million machine hours. These queries are on around 500 data streams stored in a reliable append-only distributed file system similar to the Google File System [12].

2.1 Redundancy

We have identified two kinds of redundant operations: input data scans and common sub-query computation.

Redundant I/O for scanning the input files are common in the cluster. For all the query executions in the trace, the total I/O of scanning the input files contributes to about 68% of the total I/O. The total size of input files is about 20% of the total I/O. Thus, the redundant I/O on scanning input files contributes to around 48% of the total I/O, causing a significant waste in disk bandwidth.

Redundant computation on common sub-queries is also significant. A step s in a query is defined to have a match if there exists a step s' of a previous query in the sequence, where s and s' have the same input and a common computation. Each step with a match is redundant computation because the same computation has been performed on the same data previously. In the trace, we find that 14% of the steps have a match, which contributes to around 16% of the total I/O. The I/O breakdown of redundant computation shows that 8% of the total I/O is from input steps, and the other 8% from the intermediate steps.

The overall redundancies are significant, with 56% of total I/O (48% on file input scan and 8% on intermediate steps). Since I/O continues to be a significant bottleneck for the overall performance of data center applications [1], these significant I/O redundancies contribute to a great amount of waste in total machine time. In the evaluation, our experiment demonstrates that the I/O saving of 42% results in a 40% reduction in total machine time.

2.2 Query Correlations

We found that queries are recurring and demonstrate strong correlations, even though users can only manually submit individual queries to the cluster.

Recurring queries. The queries in the trace exhibit strong temporal correlations: 75% of the queries are recurring and form around one thousand query series, each consisting of at least two queries. Inside these query series, over 67% run daily (usually with per-day input data window, e.g., requiring the log from yesterday; approximately 5% of the daily queries may involve more than one day's data); 15% are executed weekly mostly with a per-week input data window.

Data driven execution. In the cluster we study, updates are appended to streams either daily or when updates reach a predefined size threshold. Our study shows recurring queries tend to access recent updates in data streams, indicating the data-driven nature of recurring queries. Figure 1 shows the distribution of queries in query series categorized by the difference between their submission time and arrival time of the last segment they process. Around 70% of the queries fall into a window of no more than one week, and 80% no more than half a month. This shows that most query series tend to access the latest update in the target streams, and a strong indication that executions of the queries in a query series are driven by stream updates. Besides, we found that there is a gap between the data available time and the query submission time. For example, some queries with a one-day input data window are processing data that has been there for over one week. This kind of delayed submission is probably partly due to the lack of the query-series submission interface.

One micro view of three sample query series is shown in Figure 2, which shows the submission dates and the input data windows. While the submission dates are generally well aligned with the data windows, an exception is highlighted using Circle **A**. The submissions of some daily queries for the data in those three days are delayed due to a weekend.

Correlations among queries cause redundancy. In Figure 2, the input windows of the queries from query series 1 and 3 are overlapping, this results in redundant I/O scans. Examples are highlighted in Circle **B**. Redundancies show up not only across different query series, but also within a query series. In Figure 2, since queries in query series 3 have common computation on the overlapping input windows, there is often redundant computation among them, as highlighted in Circle **C**.

Stable stream characteristics. We studied the temporal stability of data streams to check the feasibility of guiding the optimization of a query based on profiling of the previous executions, especially from those in the same

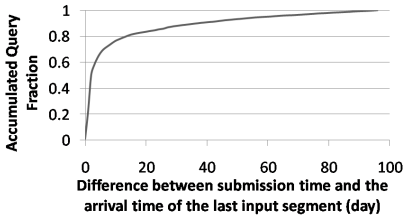


Figure 1: Accumulated query fraction.

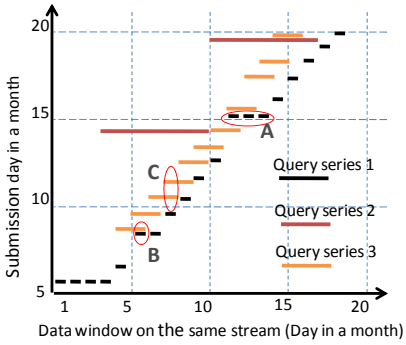


Figure 2: Sample query series on the same stream. Query series 1 and 3 consist of daily queries, with input window sizes of one and two days, respectively. Query series 2 consists of weekly queries with an input window size of seven days.

query series. We found segments of the same data stream tend to exhibit stable statistical properties.

We found the sizes of newly appended updates are stable across different days. Take the sizes of the updates to the hottest stream as an example: their coefficient of variation ($cv = \frac{stdev}{mean}$) is less than 7%.

The value distribution on each column is also stable. For example, we observed that the number of distinct values for the four most frequently used columns in the daily update is stable. The variance in the number of distinct values is small, with *cv* less than 12%.

We also examined the selectivities of the top three filters and the top three join conditions; the statistics are shown in Table 1. The *selectivity* of a filter/join is defined to be the ratio of the output size and the input size of the operation. As we can see in the table, most of the filters and the joins have stable selectivities.

Data and computation popularity. We found the

Table 1: Selectivities of the top three filters and join conditions

	Filter 1	Filter 2	Filter 3	Join 1	Join 2	Join 3
mean	0.17	0.26	8.0E-03	0.027	0.064	5E-05
stdev	0.01	0.01	1.6E-03	0.005	0.008	1E-05
cv	9%	3%	20%	17%	12%	19%

popularity of both data and query constructs conforms to *power-law distributions*, which reveals the underlying reason for significant redundancies.

Figures 3 (a,b) show the access frequencies of the data streams and their fields (or columns) during the period (normalized by the maximum frequency of the stream or field), respectively. Their distributions follow power-law distributions closely: 80% of the access are on 9.3% of the total number of data streams, and 80% of the access are on 13.9% of the total number of columns. Among the streams, we find that the frequently accessed data streams are the raw data streams shared by many users, and the infrequently accessed ones are usually for private uses only.

Similar to the power-law distribution on data popularity, we found two basic user-defined query constructs, i.e., selection conditions (filters) and custom functions, also follow power-law distributions closely. For example, the frequently used custom functions are mostly from the standard library provided in the system, and the infrequently used ones are mainly user-specific custom functions for specific processing.

In summary, we identify the clear recurring pattern in the workload, even though users can only submit individual queries manually. The processing is often driven by new updates, rather than driven by query arrivals. This motivates us to develop a new execution model for optimizations.

3 COMET DESIGN

Inspired by the trace study, we propose the BSP (Batched Stream Processing) model that explicitly captures key correlations among queries. This simple model enables vast opportunities in cross-query optimizations. We further develop Comet embracing the BSP model and with an execution model taking advantage of those optimization opportunities.

3.1 Batched Stream Processing

In the BSP model, we model data not as a static file, but as a *stream* that is periodically updated. A stream is append-only and partitioned on multiple machines. A segment is the data from a single bulk update, e.g., the daily generated log. Different segments are differentiated with their timestamps indicating their arrival times.

We further define the notion of *query series* to refer to recurrent computations on a stream, with each performed on one or more stream segments. A segment is the smallest data unit that can be accessed by a query series. A query series captures a sequence of the same computation on different sets of segments of the same stream and explicitly exposes the correlations among queries in the query series in terms of both data and computation.

This seemingly simple notion of query series brings predictability into the system, and makes cross-query optimizations tractable.

First of all, query series makes construction of a reliable cost model a possibility by leveraging the knowledge of data and computation from executions of the previous queries in the same query series. As we have already seen in the trace, the characteristics of a stream (e.g., data distribution) and custom constructs tend not to change when the stream grows over time.

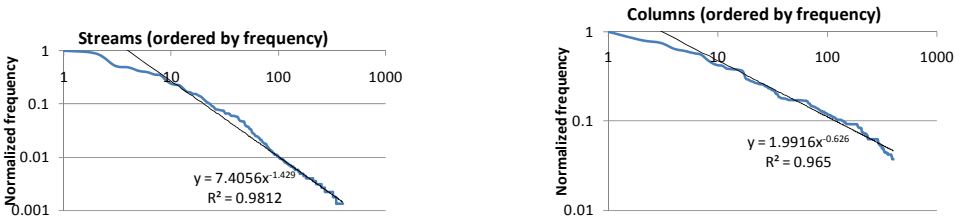
Second, due to the power-law data and computation popularity, queries in different query series are likely to share the same I/O to scan the input data and to share common intermediate steps. Those queries can be scheduled to run together as a single combined query in order to remove redundancies.

Third, a query might be decomposed into a series of smaller queries, each on a subset of input stream segments, followed by a final step of aggregating the results of the smaller queries to obtain the final result. Query decomposition enables computation be triggered by updates, since the study on the trace indicates query executions align well to new segment arrivals. Moreover, query decomposition can help uncover more opportunities for sharing. For example, if the decomposition makes all queries on the same stream process the data on aligned daily windows, there would clearly be more opportunities for sharing among queries.

3.2 Execution Model

Comet allows users to submit a query series by specifying the period and the number of recurrences of the computation. The computation of query series is triggered by updates, as the updates occur periodically. We use the following terms to capture the computation units in the execution:

- *S-query*. An S-query is a single query occurrence of a query series; it can access one or more segments on one or more streams.
- *SS-query*. Intuitively, an SS-query is a sub-computation of an S-query that can be executed when a new segment arrives. We associate with each SS-query a timestamp indicating its planned execution time. It is usually equal to the maximum timestamp of the segments it accesses: the arrival of that segment with the maximum timestamp triggers the execution of the SS-query. An S-query can be decomposed into one or more SS-queries in the normalization process (Section 4.3).
- *Jumbo-query*. A jumbo-query is a set of SS-queries with the same timestamp; that is, a jumbo query includes all SS-queries that can be executed together,



(a) Access frequency of distinct streams

(b) Access frequency of distinct columns

Figure 3: Power-law distributions on data popularity in the DISC system.

thereby leveraging any common I/O and computation among these SS-queries.

Figure 4 shows how query series are processed in Comet. When a query series is submitted, Comet normalizes it into a sequence of SS-queries and combines them with their corresponding jumbo-queries (Section 4.3). This allows Comet to align the query series based on the segments they involve. Thereafter, as with current DISC systems like DryadLINQ, Comet carries out query optimization, with an emphasis on optimizing normalized jumbo-queries (Section 4). Unlike the flow in DryadLINQ, the generated execution plans are not executed immediately; instead, arrivals of new segments start the execution of their corresponding jumbo-queries.

Comet collects statistics about data stream characteristics and operators for cost estimation. Unlike rule-based optimizations in current DISC systems, the cost estimation assesses the tradeoffs involved in various optimization choices and chooses the one with the lowest cost (Section 4.2). The statistics are stored in a *catalog*, replicated on a set of machines for reliability.

Like current DISC systems, Comet also allows users to submit ad hoc queries. Since ad hoc queries are executed on demand, Comet executes them with fewer optimizations, without normalization or combining them into the jumbo query. However, Comet does use the statistics in the catalog for optimizing ad hoc queries if they involve the same data streams and the same custom functions as those in query series. Comet further reuses its query optimization techniques to (i) find matching subqueries with results already in the system and (ii) rewrite a query to reuse available results.

4 Integration into DryadLINQ

We have integrated Comet into a recently released version of DryadLINQ. We use three sample queries to describe our integration (Figure 5), all operating on the same daily updated `log` stream (lines 2, 9, 16, and 21). The queries use a common custom function `Extractor` (lines 2, 9, 16, and 21) to retrieve rows.

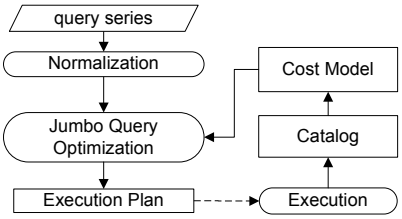


Figure 4: Comet execution flow for query series.

DryadLINQ supports the following set of operators: **P**rojection (`Select`), **S**election (`Where`), **G**rouping (`GroupBy`), **A**ggregation (e.g., `Count`), **J**oin (`Join`). The bold letters are the abbreviations of the operators for references in later figures.

4.1 Overview

DryadLINQ processes a query in the following four basic phases.[1]

(1) Translate a query into its logical plan. DryadLINQ applies logical optimizations, including early filtering and removal of redundant operators.

(2) Transform the logical plan to a physical plan with physical operators.

(3) Encapsulate the physical plan to a Dryad execution graph.

(4) Generate C# code for each *vertex* in the Dryad execution graph. Each vertex in the Dryad execution graph has several physical operator nodes, with optimizations including pipelining and removing unnecessary nodes. The vertices are deployed to different machines for distributed executions.

Figure 6 shows the logical plans for the sample queries. In DryadLINQ, physical optimizations in Phase (2) take into account the underlying distributed system configurations and data properties. Figure 6 (d) shows the corresponding physical plan for the second query. During the transformation, DryadLINQ applies local reduc-

[1] DryadLINQ also enables dynamic optimizations. We leave it out because it is not particularly relevant here.

```
// Q1: daily custom grouping on (A,B,C)
q1 = env.Extractor("log?today")
   .Select(x => new {x.A, x.B, x.C})
   .Where(x => x.A != "gb")
   .GroupBy(x => x) //grouping on (A,B,C)
   .Select(x => new {x.Key, c = x.Agg()});

// Q2: weekly histogram aggregation grouping on (A,B)
q2 = env.Extractor("log?today-6...today")
   .Select(x => new {x.A, x.B})
   .Where(x => x.A != "gb")
   .GroupBy(x => x) //grouping on (A,B)
   .Select(x => new {x.Key, a = x.Count()});

// Q3: daily join on today and yesterday's segments
q3a = env.Extractor("log?today")
   .Select(x => new {x.A, x.B, x.D})
   .Where(x => x.A != "ru")
   .GroupBy(x => x.D) //grouping on D
   .Select(x => new {x.Key, m = x.Max(y => y.B)});
q3b = env.Extractor("log?today-1")
   .Select(x => new {x.A, x.B, x.D})
   .Where(x => x.A != "ru")
   .GroupBy(x => x.D) //grouping on D
   .Select(x => new {x.Key, m = x.Max(y => y.B)});
q3 = q3a.Join(q3b, x => x.m, y => y.m, (a, b) => a);
```

Figure 5: Three sample queries `Q1, Q2`, and `Q3` in DryadLINQ. They resemble in the structure to the most popular query series in the trace.

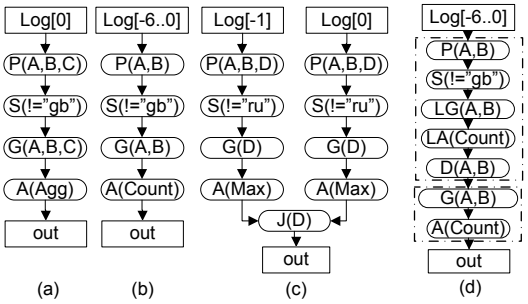


Figure 6: Logical plans for `Q1, Q2`, and `Q3` ((a),(b), and (c)), and the physical plan for `Q2` (d).

tion optimization: a local grouping (`LG(A,B)`) followed by a local aggregation (`LA(Count)`) on each machine. Then, a distributed partition phase (`D(A,B)`) shuffles the locally grouped data at the cluster level for a global grouping (`G(A,B)`) and aggregation (`A(Count)`).

We integrate Comet into DryadLINQ, enabling the execution model and its optimizations in DryadLINQ. Comet-enabled DryadLINQ allows users to submit query series as well as ad hoc queries.

The integration involves (i) adding two new phases between Phases (1) and (2) for normalization and for query matching/merging to optimize jumbo queries, (ii) adding new query rewriting rules to Phase (2) for further logical optimizations, (iii) incorporating new physical optimizations in Phase (3), and (iv) introducing a cost-model based optimization framework.

We add the step of consulting the cost model and the catalog into Phase (3), especially when the benefit and cost of some optimizations are dependent on the properties of queries and their input data streams. To allow an iterative optimization process, rather than following the pipelined process in DryadLINQ, we add a control loop between Phases (2) and (3), so that Comet can enable further optimizations after estimating the cost of the physical plans in Phase (3). Note that DryadLINQ uses run-time dynamic optimizations; for example, to figure out the number of partitions for the next stage. The cost estimation in Comet significantly reduces the needs for such optimizations.

In the following subsections, we present further details on the cost model and Comet optimizations in DryadLINQ.

4.2 Cost Model

We have implemented a simple and effective cost model for DryadLINQ. Since a precise cost model is in general hard to attain, especially for DISC systems [18], the integration of Comet alleviates the problem in two aspects. First, with Comet, DryadLINQ can take advantages of temporal correlations in the BSP model for better predictability; in particular, data properties on different segments of a data stream tend to stay the same; key properties of both data and computation are often available because the same computation has often occurred on the same data stream before. Comet collects statistics during query executions (e.g., the input and output sizes for each operator, as well as the cost of custom functions), and stores such information in the catalog for cost analysis.

Second, the integrated cost model focuses on the estimation of the total disk I/O and network I/O. Those tend to be relatively easy to estimate in DryadLINQ: for each stage, Comet can take the input size of a query and use the input/output ratio at each stage from a previous run of the same execution to estimate the amount of I/O. More importantly, we observe that I/O is the main factor that drives optimization decisions in DryadLINQ. Also, due to lack of index structures in our input data and the few number of joins, we can avoid the complications in the cost models for traditional databases [24]. Our experiments have validated the accuracy of the cost model, and its effectiveness in guiding optimization choices (Section 6).

4.3 Normalization

Comet adds a query normalization phase in DryadLINQ, prior to its logical optimization. The normalization phase converts a given DryadLINQ S-query into a sequence of SS-queries, each triggered when a segment of an input stream becomes available. This process essentially turns an S-query into an incremental computation (see [22]).

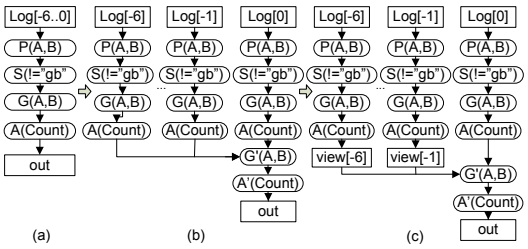


Figure 7: S-query normalization on Query Q2.(a) original logical plan for `Q2`, (b) after decomposition, (c) after adding materialized views.

In the worst case, the normalization will put the entire S-query as a single SS-query.

Figure 7 depicts the normalization process for an S-query of the second sample query series. As the S-query involves one week's data (Figure 7 (a)), we split the input node into seven nodes, each indicating one daily segment (Figure 7 (b)). Comet then explores the paths starting from those nodes, examines each operator at each level from the top down, and splits the operator when it is appropriate based on decomposability of that operator.

Decomposability of an operator indicates whether it is feasible to split its computation. Most of the operators, such as projections and selections, are easily decomposed. Some require an extra phase at the end to produce correct results; for example, an extra merge node `A'(Count)` is added for generating the final weekly histogram from the daily ones (Figure 7 (c)). There are also others, such as aggregations with some customized functions, which cannot be decomposed because we cannot easily understand how to make the decomposed plan produce the same output as the original one.

As with DryadLINQ, Comet must infer the type of parameters in the newly constructed expression tree. Because LINQ uses strongly typed lambda expressions, this inference process could be tedious if the new expression tree is different from the original one. For example, when decomposing Query Q2, the first seven has the same structure as the original query and therefore its expression tree is easy to construct. The final one that merges the results from the first seven has a different structure.

We arrive at Figure 7 (b) after the operators are decomposed. Comet further splits this plan into several independent SS-queries: it adds an output node to each of the last decomposed operator `A(Count)` in the previous 6 days. The inserted output node is considered as a *materialized view* [2], and the sub-graph that ends at this output node is considered an SS-query. The rest of the plan is then another SS-query to be executed on the final day. The SS-query not only takes the final segment as the input, but also uses materialized views from previous SS-queries.

After getting all SS-queries from all query series, Comet aligns them and constructs a jumbo-query for all the SS-qeuries with the same timestamp, as shown in Figure 8 (a), for further optimizations. Through normalization, redundancies across queries are exposed to the later logical and physical optimizations of Comet-enabled DryadLINQ.

4.4 Logical Optimization

Comet enables new logical optimizations including shared computation and reused views for DryadLINQ to remove redundancies in the logical representation of jumbo queries. These techniques are rooted in logical query optimizations in relational databases [23, 29].

Shared computation. While current DryadLINQ can identify shared computation on common expressions across SS-queries inside a jumbo-query, its rule-based optimization process limits sharing opportunities. To enable more sharing opportunities, Comet employs operator reordering. For example, as shown in Figure 8 (b), Comet changes the order of `P(A,B,C)` and `S(!="gb")`, as well as `P(A,B)` and `S(!="gb")` in Figure 8 (a), so that we can combine the selection operators because their filter conditions are exactly the same.

Operator reordering generates multiple possible plans, and the cost model evaluates which one is better based on the statistics about selectivities stored in the catalog. Considering two branches that have two different selection operators followed by two grouping operators with the same grouping keys. We can swap the selection operators and the grouping operators, so as to do the grouping operation only once. However, if the two selections can reduce the input data size dramatically, this optimization actually hurts the overall performance. Note that current DryadLINQ chooses the later case according to its rule. As our evaluation shows that the rule is not always true: a wrong decision may lead to up to 30% penalty in I/O.

Reused view across jumbo-queries. Redundant computation can also occur across jumbo queries. This happens when two jumbo queries operate on overlapping input segments. For instance, the output of `A(Max)` in Figure 8 (a) can be reused in the execution of the next jumbo query when a new segment arrives. Comet stores the outputs as materialized views.

Comet further specifies the co-location feature for partitioned views. Co-location is an important consideration because it could reduce network traffic significantly. While co-location of intermediate results within a jumbo-query is well taken care of by the compiler [16, 18], co-location of the results across jumbo queries however needs special care.

One example of partitioned views with co-location is shown in Figure 8 (a), where a later join operator `J(D)`

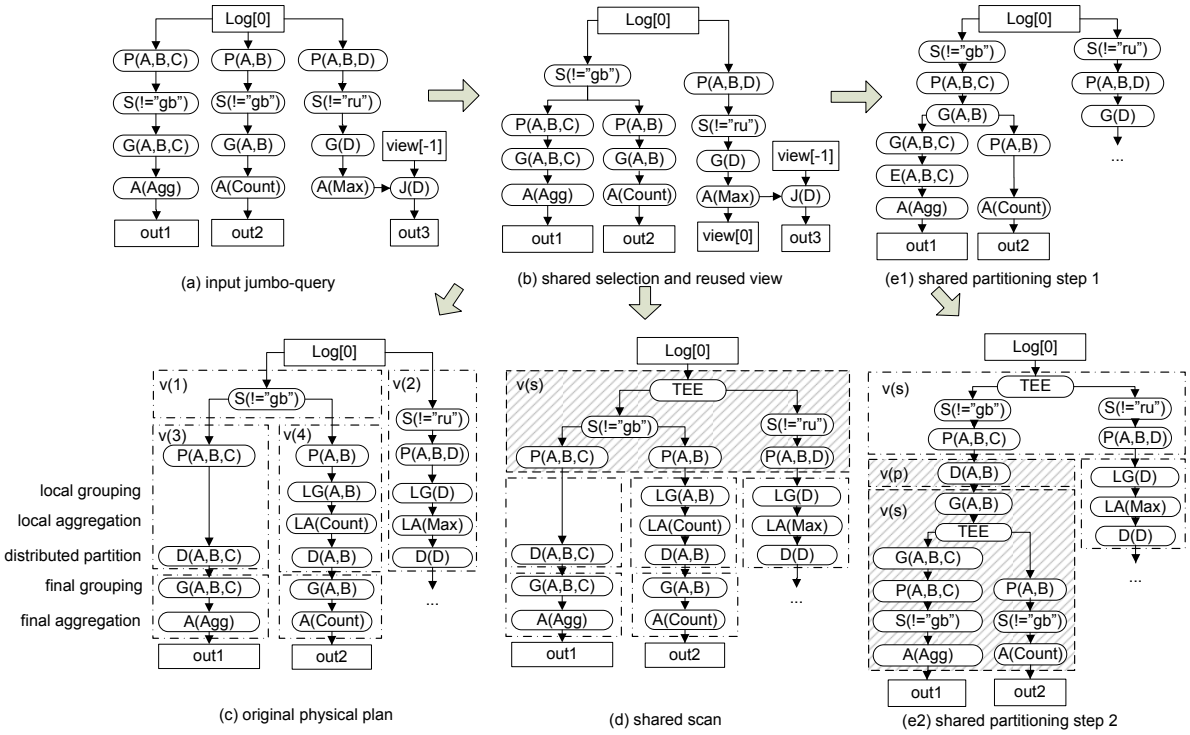


Figure 8: Logical and physical optimizations of Comet.

takes the view `view[-1]` from yesterday and the aggregation result today (`A(Max)`), both of them partitioned with the same `D` field. Comet partitions `view[-1]` and `A(Max)` in the same way and co-locates the corresponding partitions, so that the join can be done locally on each partition. Another example is shown in Figure 7 (c), where the final SS-query might have to merge the results from all the previous SS-queries. The trace study shows that around 8% of the queries whose inputs last for more than one day can gain benefits from co-location.

4.5 Physical Optimization

Comet enables new physical optimizations for efficient data sharing in DryadLINQ. Sharing opportunities at the physical level manifest themselves as branches in a logical plan after logical optimizations (e.g., Figure 8 (b).) There are two types of branches, one enables *shared scan* [1] and the other *shared partitioning*. The differences lie in whether the branching point corresponds to local data transfers or network data transfers.

Shared scan. An example of shared scan is shown in Figure 8 (b): `S(!="gb")` and `S(!="ru")` share the same input node `Log[0]`. Current DryadLINQ tends to separates the nodes into different vertices whenever it encounters branching. That is, current DryadLINQ puts the two nodes into different vertices (see Figure 8 (c)), which results in two scans on the same input segment (to `v(1)` and `v(2)`).

To enable efficient shared scan, Comet puts all branching nodes in a Dryad vertex. Comet implements a new physical operator `TEE` for combining, which connects one input stream provider with multiple concurrent consumers. For example, Comet applies the shared scan optimization to put `S(!="gb")` and `S(!="ru")` inside one vertex (`v(s)` in Figure 8 (d)), so that they can cooperate with each other to reduce the number of scans on the input segment to only one.

Shared partitioning. An example of shared partitioning can be found in Figure 8 (b): the output of `S(!="gb")` is shuffled across machines twice by `G(A,B,C)` and `G(A,B)`. Note that the two grouping operators have a common prefix `(A,B)`. Comet partially shares the two grouping operators by pushing down `P(A,B,C)` and `P(A,B)` to eliminate redundant data shuffling (Figure 8 (e1)). (Note that this is a logical rewrite triggered at a later stage.) Then, it transforms this logical plan to a physical one (Figure 8 (e2)), translates the shared grouping to a shared distributed partitioning `D(A,B)` in vertex `v(p)`), and further enables the shared scan optimization by grouping the further operators for the two SS-queries into one vertex (`v(s)`).

There are tensions between single-query optimizations and Comet optimizations, which exploit sharing among queries. For example, DryadLINQ uses early aggregations to reduce the I/O in later stages, but this could eliminate certain opportunities for removing redundan-

cies across queries. More concretely, two different physical execution plans for the same jumbo query are shown in Figure 8 (d) and Figure 8 (e2). The former applies the early aggregation optimization (`LA(Count)`), which can usually reduce data sizes for a later distributed partitioning phase across the network incurred on individual SS-queries. The latter applies the shared partitioning optimization to reduce the redundant data shuffling across the SS-queries. Unlike DryadLINQ always choosing the former plan, Comet relies on the cost model to predict which one is better.

5 COMET SIMULATOR

We implement a trace-driven simulator that is capable of estimating savings due to Comet optimizations. Taking a trace from a real workload as input, the simulator first categorizes queries into two kinds, i.e., ad hoc queries and recurring queries, and then processes the queries accordingly.

As for query series, the simulator maintains global logical and physical plans, representing all the jumbo queries that have been processed. The simulator tries to match the query plan of a jumbo query against the global query plan and calculates the benefits of each optimization technique. For example, if the query plan exactly matches a path in the global execution plan, we add the total I/O of the query plan to the savings from logical optimizations. In particular, the Comet simulator simulates the following aspects in Comet.

– *Simulating query normalization.* The simulator normalizes queries into SS-queries and evenly distributes the total I/O costs of the original query to the SS-queries.

– *Simulating logical optimizations.* The simulator removes redundancies in jumbo queries. The materialization cost of creating materialized views is counted. The cost involves writing two extra copies of the data, replicating the data twice in the distributed file system for reliability.

– *Simulating physical optimizations.* The simulator optimistically estimates the benefits of shared scan and shared partitioning: the cost of only one input scan or partitioning is counted.

The Comet simulator also supports simulating the execution of ad hoc queries. Due to the unpredictability of ad hoc queries, the simulator considers whether an ad hoc query can reuse the views generated from the previous execution of query series.

Finally, the Comet simulator outputs the I/O cost of the simulated workload. We have experimentally validated the accuracy of the simulation in Section 6.2.

6 EXPERIMENTS

We perform two sets of experiments to evaluate Comet. The first set of experiments is on a real deployment of Comet on a cluster of 40 machines using a micro benchmark. This micro benchmark is to reveal the micro-level details of Comet integrated into DryadLINQ. The second set of experiments applies the simulator on the entire real-world trace reported in Section 2 to show the global effectiveness of Comet, in comparison with two existing multi-query optimization approaches [1, 18] in DISC systems.

We mainly focus on system throughput, and use two metrics in the micro benchmark: the *total machine time* and the *total I/O*, where the total machine time is the aggregated total amount of time (in seconds) spent on the entire query on all the machines involved, and the total I/O is the number of bytes (in GB) in the disk and network I/O during the execution. We have run each experiment five times. Variances among different runs are small, and we report the averages. As for cost estimation, we use total I/O as the main metric in the simulation.

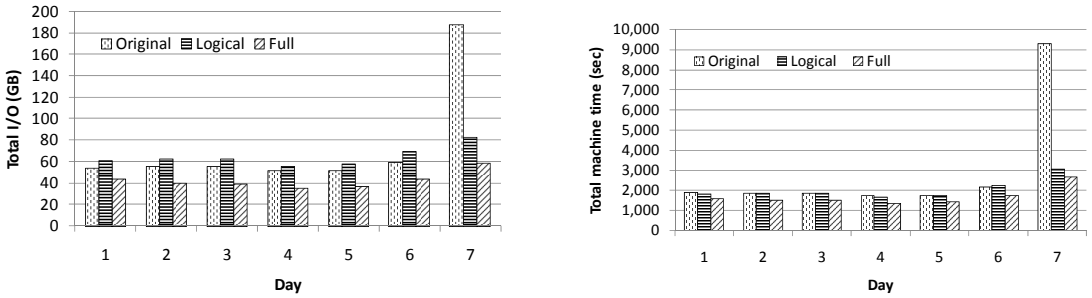
The optimal execution plans for jumbo queries are automatically generated according to the cost model in Comet. To evaluate the separate benefits of individual optimization techniques, we manually enable/disable certain optimizations. Overall, we define three optimization configurations: *Original*, under which queries are executed without Comet optimizations; *Logical*, which adopts only query normalization and logical optimizations; and *Full*, which includes query normalization, logical and physical optimizations.

6.1 Micro Benchmark Study

The micro benchmark study evaluates the overall benefit as well as the cost model and choices in the jumbo query optimization, e.g., shared scan and shared partitioning.

We construct a micro benchmark, which consists of the three query series in Figure 5. Their structure resembles that of the most popular query series we have found in the query trace of the real workload. The dataset contains per-day segments of a real stream from the same workload. The average size of the per-day segments is around 2 TB. We projected the five most popular columns (denoted as A–E) into a stream. Each segment is evenly partitioned and stored to the machines in the cluster. The average size of each segment is around 16 GB. Column A has a relatively small number of distinct values with an uneven distribution, Column B follows the zip-f distribution, and Columns C, D, and E are nearly evenly distributed.

The final execution plans for the jumbo-query based on the three query series and the given dataset are as follows: *Original* uses a normal DryadLINQ generated ex-



(a) Total I/O

(b) Total machine time

Figure 9: The effectiveness of Comet with a detailed breakdown.

ecution plan, *Logical* uses the plan in Figure 8 (c), and *Full* uses the plan in Figure 8 (e2) (we will discuss why Comet did not select Figure 8 (d) later.) All the experiments on the micro benchmark are performed on a cluster of 40 machines, each with a dual 2GHz Intel Xeon CPU, 8 GB memory and two 1TB SATA disks, connected with 1 Gb Ethernet.

Overall effectiveness. We run the three query series against the data of several weeks, and we repeated the experiment under the three optimization configurations. Figure 9 (a) shows the total I/O of one week in a steady execution, denoted as day 1 to day 7. In *Original*, Q1 and Q3 are daily executed from day 1 to day 7, and Q2 is executed on day 7 only. *Original* has a sharp spike in the total I/O on day 7 due to Q2.

For both *Logical* and *Full*, queries are normalized into SS-queries, which are aligned with the per-day stream updates. From day 1 to day 6, both *Logical* and *Full* are stable, as they repeat the same jumbo-query. On day 7, they perform an additional final grouping operation on the seven materialized views generated in previous jumbo-queries for the second query series, thereby incurring higher total I/O. Figure 9 (b) shows the total machine time, which is consistent with the results on the total I/O.

With logical optimization, Comet reduces the total machine time by 30.5%, and the total I/O by 12.3%. With both logical and physical optimizations, Comet reduces the total machine time by 42.0%, and the total I/O by 42.3%. Besides, since our optimization divides the execution for Q3 into seven days, *Logical* and *Full* have a more balanced load compared to *Original*.

Cost Model Accuracy. We evaluate the accuracy of our cost model by comparing the total I/O numbers from the practical runs above and from estimation based on the statistics from the previous runs, i.e., we use the statistics from the previous execution and the input size of the current one to estimate the cost of the current execution. The experiment was done twice under the *Original* and *Full* configurations. Figure 10 shows the result: the estimated total I/O follows the actual total I/O closely, which validates the accuracy of our cost model.

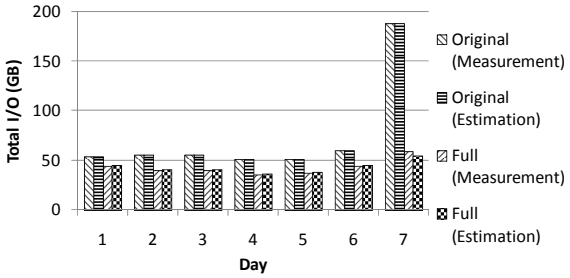


Figure 10: Estimated and measured total I/O from day 1 to day 7.

Shared Scan. To investigate the appropriate number of branches to be combined in one shared scan vertex (Section 4.5), we combine 16 SS-queries from the first query series together with a varying number of branches in one shared scan vertex (1, 2, 4, 8, and 16). For example, we split the 16 SS-queries into 4 vertices when the number of branches in one vertex is 4. Figure 11 shows the total machine time with different selectivities (by changing the filtering condition). The results show that it is beneficial to enable shared scan with a large number of branches when the selectivity is low: this is the case for most queries in our trace. When the filtering ratio is high, the appropriate number of queries per shared scan vertex should be small. Our cost model can guide this decision making.

Early filtering versus shared partitioning. As discussed in Section 4.5, pushing down a selection operator for shared partitioning is not always profitable. We investigate how the total I/O varies with the selectivity of the selection operators in the first query series (by changing the filtering conditions.) The experiment was done twice on two instances of the first query series: the

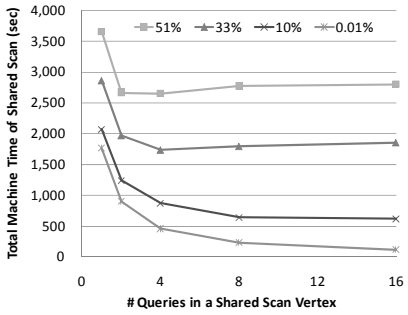


Figure 11: Total machine time of the shared scan.

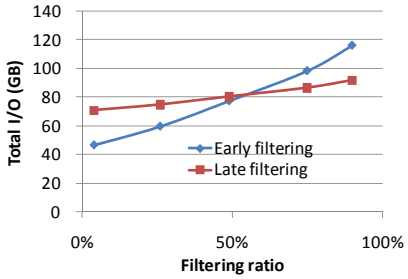


Figure 12: Performance on running two instances of `Q1`.

first instance applies early filtering without shared partitioning; the second instance applies late filtering with shared partitioning. Figure 12 shows the total I/O numbers for the two cases with the varying selectivity: As the selectivity increases, the benefit of shared partitioning increases. When the selectivity is larger than 50%, late filtering is preferable. Comet decides early filtering or shared partitioning using cost estimation according to selectivity.

Early aggregation versus shared partitioning. Recall that Comet selected Figure 8 (e2) (w/ shared partitioning) instead of Figure 8 (d) (w/ early aggregation) as our *Full* optimization benchmark. The reasons are as follows. The `Count` early aggregation applied to the second query series can save only 0.2 GB of the network I/O, due to the distribution of the partitioning key `A,B` for the aggregation. Meanwhile, the shared partitioning between the first and second query series can save 0.9 GB network I/O, which is much more profitable.

Co-location of partitioned views. We also evaluate the impact of co-location between two partitioned views that are to be joined as in the third query series. We run the experiments without and with co-locating the two views, and found that the total I/O is reduced from 191.6 GB to 175.2 GB, which has performance gain of 8.6%.

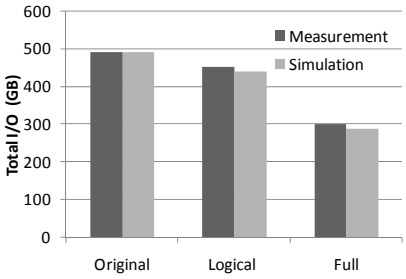


Figure 13: Simulation validation on micro benchmark.

6.2 Global Effectiveness Study

The simulator offers the capability of estimating the effectiveness with a real trace from a production system. In this study, we use the trace described in Section 2 and report the simulation results. The average optimization time under the *Full* configuration on the trace of each day is consistently under one minute, leading to an average of under one second for each query series. This overhead is negligible in practice, compared with typical query execution time in the real cluster.

Simulation validation. We first evaluate the accuracy of our simulation. Figure 13 shows the total I/O of the micro benchmark reported in the above experiments and our simulation, with the three optimization configurations. We found that the maximum deviation is approximately 5% for the *Full* configuration, which validates the credibility of our simulation.

Overall effectiveness. We run the simulator on the entire trace under the two optimization configurations. By default, the optimization configurations are with query normalization. To study the effectiveness of normalization, we also simulate them without normalization. Figure 14 shows the simulated results under these four configurations. We can see that the *Full* optimization with normalization can reduce the total I/O to 48%, which is a significant cost saving. The *Full* optimization without normalization can also reduce the total I/O to 58%, which means that the normalization contributes 10% of the cost saving; this is consistent with the fact that over 67% of the query series in our trace are daily executed that need no query normalization. For the remaining two *Logical* configurations, we can still get 15% to 16% performance gain in terms of the total I/O.

Performance gain breakdown. A closer look at the savings from logical optimizations reveals that around 76% and 22% of their savings come from the extraction and aggregation operations, with around 2% due to the shared computation from other operations. As for physical optimizations, over 97% of all the physical optimization savings come from shared scan, with the remaining due to shared partitioning.

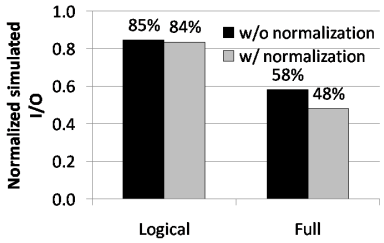


Figure 14: Total I/O in simulation with and without query normalization.

Table 2: Normalized simulated I/O of different optimization approaches

Approaches	Normalized simulated I/O
Original	100%
MV	85%
SM	80%
MV+SM	75%
Comet	48%

We further studied the amount of daily I/O with *Full* optimization. During our observed period, the amounts of daily I/O with *Full* are between 35% and 98% of those with *Original*, indicating our optimization techniques reduce the total I/O for every day.

Performance gain on ad hoc queries. We further look at the component of the ad hoc queries in our evaluation. Those ad hoc queries account for 30% of the total I/O in the original trace, but the ratio goes up to 61% after the optimizations because our optimizations are more effective to recurring queries. Ad hoc queries also benefit if some of their computation has already been performed previously. Our results show a saving of 2% in terms of the total I/O for the ad hoc queries.

Comparison with existing approaches. Finally, we implemented two existing complementary multi-query optimization approaches in the simulator, one based on materialized views [18] for caching (denoted as *MV*), and the other scheduling input scans [1] (denoted as *SM*). *MV* identifies the common computation from the workload in the previous day, and stores the results for the common computation of the current day for reuse. *SM* considers shared scans among queries in the query waiting queue. Compared with Comet, these two approaches do not have query normalization or global optimizations on shared input scans.

Table 2 shows the normalized I/O of simulating different multi-query optimizations, where *MV+SM* denotes the result of applying both *MV* and *SM* approaches to the system. Comet is more effective than *MV+SM*, with 36% less I/O.

6.3 Summary

The evaluations on the micro benchmark show that the optimizations in Comet reduce 42.0% of total machine time and 42.3% of total I/O. The simulations show that Comet reduce over 50% of the total I/O, and has 36% less I/O than the combined optimization of the two existing multi-query optimization approaches in DISC systems [1, 18]. Reducing the total I/O by one half is likely to double the capacity of the system, which can save a great deal of the investment on the data center.

7 Related Work

Comet builds on prior work in both data intensive distributed systems and databases systems, especially those on query optimizations.

7.1 Large-scale Data Processing Systems

The need for large-scale data processing has given rise to a series of new distributed systems. The state-of-the-art execution engines, such as MapReduce [8], Dryad [16], and Hadoop [14] provide scalable and fault-tolerant execution of individual data analysis tasks. More recently, high-level languages, such as Sawzall [20], Pig Latin [19], DryadLINQ [28], and SCOPE [6] introduce high-level programming languages, often with certain SQL flavors, to facilitate specification of intended computation. All these systems adopt the batched processing model and treat queries individually. A set of optimization techniques, such as early aggregation (or local reduction), have been proposed and incorporated into those systems, almost exclusively for optimizing individual queries.

Olston et al. [18] recognizes the relevance of database optimization techniques and proposes a rule-based approach for both single- and multi-query optimizations in batch processing. They further propose shared scans of large data files to improve I/O performance [1]. The work focuses on a theoretical analysis, with no report of real implementations or evaluations. The adoption of the BSP model does help Comet address the challenges that were considered difficult: the BSP model allows a natural alignment of multiple queries to enable shared scans and makes a simple and accurate cost model feasible. The techniques in [18, 1] are useful for Comet to handle ad hoc queries.

Our previous work [15] presents the preliminary study on the trace and outlines some research opportunities. This work extends the previous study with an emphasis on query correlations as well as data and computation popularity, and realizes the research opportunities through the proposal of the BSP model and the integration into DryadLINQ.

7.2 Database Optimizations

Many core ideas in Comet optimizations can find their origins in query processing techniques in database systems [9, 13], both in batch processing [9, 23, 25] and in stream processing [3].

As with the stream processing model [3], computation in the BSP model is triggered by new updates to data streams, but without resource and timing constraints normally associated with stream processing; as with the batch processing model, each query in a query series is a batch job, but the computation is recurring, as it is triggered by a (bulk) update to data streams.

Batch processing. There is a large body of research on query optimizations for batch processing in traditional (parallel) databases [9]. Shared-nothing database systems like Gamma [10] and Bubba [5] focus mainly on parallelizing a single query. As for multiple query optimizations, materialized views [2, 23] are an effective mechanism in exploiting the result of common subexpressions within a single query or among multiple queries. Zhou et al. improves the view matching opportunities on similar subexpressions [29]. In Comet, persistent outputs registered in a catalog are materialized views, without complicated and usually expensive view maintenance in databases [4]. Additionally, by combining queries into a jumbo query, the results of most common expressions do not need to be materialized. Zukowski et al. [30] enhanced the existing I/O scheduling policies for concurrent disk scans.

Stream processing. Stream processing systems such as STREAM [27], OpenCQ [17], and NiagaraCQ [7] usually process real-time and continuous data streams. Due to resource and time constraints, stream data are usually not stored persistently. Continuous queries run on a stream for a period of time, and return new results as new data arrives. Query processing algorithms for incremental computation [26, 17] and for identifying common subqueries among continuous queries [7] are proposed to process streams efficiently.

8 DISCUSSIONS

The BSP Model and Ad Hoc Queries. Comet's design targets the BSP model, but can easily accommodate ad hoc queries. In fact, we expect that for any DISC systems the workload will consists of those conforming to the BSP model and those ad hoc queries that do not. Many optimization techniques in Comet can benefit ad hoc queries as well, as our simulation indicates. Clearly, ad hoc queries cannot take full advantages of the optimizations in Comet: because an ad hoc query is triggered upon submission, it cannot be easily aligned with other queries for shared scans; because an ad hoc query is non-recurring, the cost model might be less accurate.

The co-existence of the BSP queries and the ad hoc queries also impose challenges on other parts of the system. Ad hoc queries are likely to be significantly smaller than jumbo queries. Fairness in scheduling thus becomes crucial for providing a reasonable service to ad hoc queries.

The BSP Model and its Impact on the Underlying System. Comet can also benefit from better support from the underlying execution engines. Currently, jumbo queries that Comet creates are given to the underlying system as a single job: the information that the job contains multiple queries is lost. This makes it hard to achieve the fairness among queries in a job and for preventing failures of individual queries from aborting other queries in the same job.

There is also a tension between maximizing sharing and enabling parallel executions. For example, to get the benefits of shared scan and shared partitioning, multiple queries are now scheduled to run on the same machines concurrently. While this optimizes the overall throughput and improves resource utilizations, it might create hotspots in some part of the system, with idle resources elsewhere. As for the power-law popularity on data accesses, a distributed storage system must balance allocation of cold and hot data; this will help alleviate the tension in Comet.

Declarative Operators and Imperative Custom Functions. The combination of declarative operators and imperative custom functions in DryadLINQ might appear to be a prefect choice for expressiveness, ease of programming, and flexibility. But the effect of pollution from those imperative custom functions is particularly alarming, especially for some of the seeming natural optimizations we would like to perform. It seems to echo some of criticisms from the database community [11]. Some way of constraining that flexibility seems desirable.

The issue has already surfaced in the original DryadLINQ system, as hinted by its authors. Optimizations such as early aggregations become hard with custom aggregation functions. The custom functions also make it hard to propagate the data properties that are important for optimizations. One proposal is to let users annotate the functions. Comet faces a more significant challenge: because Comet optimizations are often across queries from different users, users might not be aware of such optimizations to help. We believe that a combination of automatic program analysis and tasteful constraints on the custom functions might help address the issues.

9 CONCLUSIONS

With the increasing uses of distributed systems in large-scale data processing, we envision the inevitable convergence of database systems and distributed systems in this

context, which will bring a set of new challenges and opportunities in performance and resource optimization. Motivated by the observations from a real system, Comet is a step towards that convergence, embracing an execution model driven by arrivals of data updates, and making cross-query optimization tractable. We hope that insights from Comet, especially on execution models and optimization techniques, and their interplay, could help inspire and influence future research in this emerging and increasingly important area.

REFERENCES

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, 2008.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM PODS*, 2002.

[4] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, 1986.

[5] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, 1990.

[6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2), 2008.

[7] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, 2000.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

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

[10] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[11] D. J. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 1, 2008.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[14] Hadoop. *http://hadoop.apache.org/*.

[15] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[17] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11:610–628, 1999.

[18] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX ATC*, 2008.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *ACM SIGMOD*, 2008.

[20] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4), 2005.

[21] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *HotCloud*, 2009.

[22] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[23] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2), 2000.

[24] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD*, 1979.

[25] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[26] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *ACM SIGMOD*, 1992.

[27] The Stanford STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin, 26(1)*, 2003.

[28] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[29] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *ACM SIGMOD*, 2007.

[30] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *VLDB*, 2007.