Comet: Batched Stream Processing for Data Intensive Distributed Computing

Bingsheng He
[†]Microsoft Research Asia

Mao Yang Zhenyu Guo
Microsoft Research Asia

Rishan Chen[†‡]
[‡]Peking University

Bing Su
Microsoft Research Asia

Wei Lin
Microsoft

Lidong Zhou
Microsoft Research Asia

Abstract

Batched stream processing is a new distributed data processing paradigm that models recurring batch computations on incrementally bulk-appended data streams. The model is inspired by our empirical study on a trace from a large-scale production data-processing cluster; it allows a set of effective query optimizations that are not possible in a traditional batch processing model.

We have developed a query processing system called Comet that embraces batched stream processing and integrates with DryadLINQ. We used two complementary methods to evaluate the effectiveness of optimizations that Comet enables. First, a prototype system deployed on a 40-node cluster shows an I/O reduction of over 40% using our benchmark. Second, when applied to a real production trace covering over 19 million machine-hours, our simulator shows an estimated I/O saving of over 50%.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed systems—*Distributed databases*; H.2.4 [**Database management**]: Systems—*Distributed databases, Parallel databases, Query processing*

General Terms

Measurement, Performance, Management

Keywords

Data-intensive scalable computing, resource management, batched stream processing, query series

1. INTRODUCTION

Data intensive scalable computing (DISC) systems, such as MapReduce/Sawzall [10, 21], Dryad/DryadLINQ [18, 30], Hadoop/Pig [16, 20] and SCOPE [6], have unanimously embraced a *batch* processing model, where each *query* specifies computation on a large bulk of data. These systems tend to process queries individually. In reality, we face the challenging problem of executing a large number of complicated queries on a large amount of data every day across thousands of servers. Optimization of query executions is essential for effective resource utilization and high throughput.

We have examined a 3-month trace from a production data-processing cluster. This trace captures a workload that consists of 13 thousands queries costing a total of 19 million machine-hours. Our study on the trace focuses on network and disk I/O because they are major performance factors in data intensive computation [1, 19, 17, 22]. The study reveals that system efficiency is far from ideal. For example, we find that over 50% of total I/O is spent on repetitive input-data scans and on repetitive computations across different queries. Significant I/O redundancies cause significant waste in I/O bandwidth, which often translates into significantly reduced system throughput.

Our study further reveals 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 computations 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 computations. For example, one data stream might store web search logs and is appended daily with new entries. A query might be issued daily to retrieve the top ten hottest keywords from the search logs over a sliding window of the last 7 days. These daily queries have clear temporal correlations. Another set of daily queries might probe the geographical distribution of search entries in the same 7-day window, thereby exhibiting spatial correlations with the first set of queries.

To expose temporal correlations among queries, we introduce *Batched Stream Processing* (BSP) to model recurring (batch) computations on incrementally bulk-appended data streams, which is a dominant pattern that we observed in the studied trace. Recurring computations on the same data stream form a *query series*. An example query series consists of the daily queries for the hottest keywords in the last 7 days, as described earlier. With query series, an execution of an earlier query in a query series could help the execution of later queries in the same query series. First, it could preserve intermediate results that are needed by later

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0036-0/10/06 ...$10.00.

queries; for example, for the hottest-keyword query series, it might be beneficial to create a daily keyword count because it will be used by the next 6 days of queries in the same query series. Those intermediate results resemble *materialized views* [2] in database systems. Second, profiling the execution of an earlier query could help guide optimizations of later queries in the same query series.

Later queries in a query series are driven by bulk updates to input data streams, rather than being triggered by query submission from users. This has significant implications. Queries from multiple query series operating on the same input data stream can now be aligned to execute together when new bulk updates occur. This maximizes opportunities to remove redundant computations or I/O across those queries; such redundancy arises from spatial correlations among those queries. With the Batched Stream Processing model, traditional database optimization techniques, especially those for continuous queries [27] and multiple queries [26], become relevant and applicable to DISC systems.

We have built *Comet* as a system to support Batched Stream Processing: Comet allows users to submit query series and implements a set of global optimizations that take advantage of the notion of query series. Query execution in Comet is triggered by arrivals of new bulk updates to streams. A query is decomposed into a number of sub-queries, each of which is performed on a new bulk update. Comet aligns sub-queries from different query series into a single *jumbo query* and optimizes the jumbo query to remove redundancies and improve performance.

We have integrated Comet into DryadLINQ [30] and enabled a set of query optimizations that are not available in DryadLINQ. We used two complementary methods to evaluate the effectiveness of Comet. We have built a simulator to estimate the I/O cost of a DISC system, both with and without Comet optimizations. Our simulation of the production trace covering over 19 million machine-hours shows a reduction over 50% with Comet optimizations. We further implemented a prototype system on top of DryadLINQ and deployed the system on a 40-machine cluster. The prototype was used to validate our simulation results. When running a micro-benchmark, our prototype shows that Comet optimizations cuts the total I/O cost by over 40%.

The rest of the paper is organized as follows. Section 2 describes our empirical study on a real-world workload from a production system. We present the Comet design in Section 3 and the integration into DryadLINQ in Sections 4. Section 5 presents experimental results. We review the related work in Section 6, with a discussion in Section 7. Finally, we conclude in Section 8.

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 , such as search query logs and click logs. The trace documents the information related to executions of all query jobs in the system. The information includes the 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 that take approximately 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 [14]. The data streams are appended regularly in bulk.

2.1 Redundancy

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

Redundant I/O for scanning input files are common in the trace. For all query executions in the trace, the total I/O of scanning 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, potentially causing a significant waste in disk bandwidth.

Redundant computations on common sub-queries are 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 a 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 computations shows that 8% of the total I/O is from input steps, and the other 8% from intermediate steps.

The overall redundancies are significant, with 56% of total I/O (48% on input file 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.

2.2 Query Correlations

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

Recurring queries. 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 data spanning over multiple days); 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, indicating the data-driven nature. 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 that 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 a strong indication that executions of the queries in a query series are driven by stream updates. In addition, we have 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 a query-series submission interface.

One micro view of three sample query series is shown in

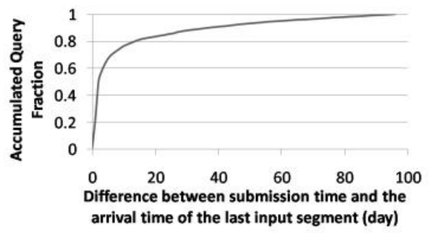
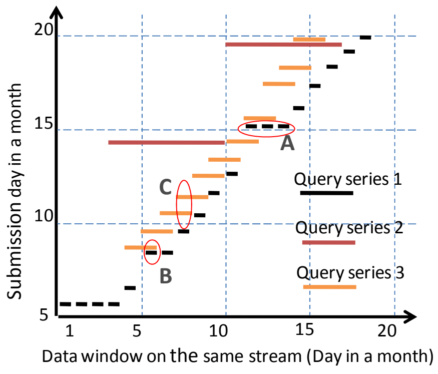


Figure 1: Accumulated query fraction.



Submission day in a month
Data window on the same stream (Day in a month)
A
B
C
Query series 1
Query series 2
Query series 3

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.

Figure 2. The figure 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, input windows of the queries from query series 1 and 3 are overlapping, resulting 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 computations on overlapping input windows, there is often redundant computations (Circle **C**).

Load distribution. Our study also reveals noticeable temporal load imbalance in the system. Figure 3 shows the normalized total I/O for all query executions per day in a month. The total I/O fluctuates with a certain pattern: the total machine time on weekdays is on average 50% higher than on weekends. This is partly because query executions are triggered upon user submissions, which tend to happen during weekdays. Queries with per-week (or per-month) data updates tend to execute at the beginning of every week (or every month), when updates become available. Another example is highlighted using Circle **A** in Figure 2. Some daily queries for the data in those three days are delayed due to a weekend. These result in a burst of activity after that weekend, contributing to load imbalance.

2.3 Data Stream Properties

We have studied the temporal stability of data streams to check the feasibility of guiding the optimization of a query

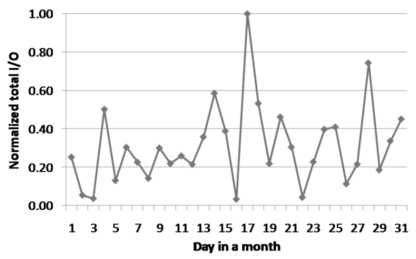


Figure 3: Daily total I/O in a month

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%

based on profiling of the previous executions, especially from those in the same query series.

We have found data distributions of newly appended updates are stable across different days. 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 a cv (coefficient of variation, $cv = \frac{stdev}{mean}$) of 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 is defined to be the ratio of the output size and the input size of the operation. The join selectivity is defined as $\frac{O}{I_1 \times I_2}$, where O is the number of entries in the result and I_1 and I_2 are the numbers of entries in the two inputs of the join. As we can see in the table, most of the filters and the joins have stable selectivities.

Take Filter 1 as one example. Figure 4 shows the normalized input and output data sizes in a series of recurring queries. For each query, the output data is obtained through applying Filter 1 on the input data. The output data size is clearly correlated with the input data size, thereby providing excellent hints on the selectivity of later recurring queries from the previous ones.

Finally, we found that the frequency of data stream accesses conforms to *power-law distributions* closely: 80% of the accesses are on 9.3% of the data streams. This distribution reveals spatial correlations among queries. 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.

In summary, our study of the trace reveals strong tempo-

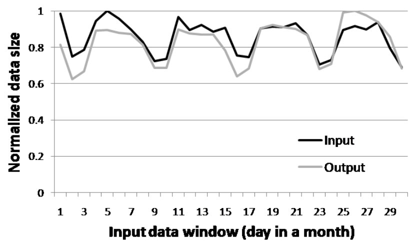


Figure 4: Normalized input and output data sizes for Filter 1 in a series of recurring queries

ral and spatial correlations among queries; those correlations lead to significant I/O redundancies and temporal load imbalance. There is a clear indication that queries are mostly driven by new updates. Recurring queries are expected to exhibit similar behavior because of the stability observed on data stream properties: this lays the foundation for optimizing recurring queries based on the profiling of an earlier execution. All these results argue for the batched stream processing model, as well as hinting at the potential benefits of such a model.

3. COMET DESIGN

Our study on the production trace indicates that a significant portion of queries follow the Batched Stream Processing model, where source input data are modeled as periodically appended *streams* with recurring queries triggered upon bulk appends to these streams.

Each single bulk update creates a *segment* of the data stream; different segments are differentiated with their timestamps that indicate their arrival times. Recurring computations form a query series, where each query instance in a query series is triggered when new segment(s) are appended.

In the batched stream processing model, users can submit query series that explicitly convey temporary correlations among individual queries; while in a traditional batch processing system these queries would have to be submitted separately. This seemingly simple notion of query series enables a set of new optimizations that are not available or hard to implement in current batch-processing systems.

With query series, execution of an earlier query in a query series is aware of future query executions in the same query series, thereby offering opportunities for optimizations. First, an execution of an earlier query could piggyback statistical properties of input data streams or intermediate data; such statistical information could guide effective optimizations of later queries. As we have already seen in the empirical study, important statistical properties such as the data distributions of stream and filter selectivity tend to be stable as a data stream grows over time. Previous executions are also effective in estimating the cost of custom functions, which are an important part of data processing queries. Such estimation would have been difficult otherwise. Second, in cases where consecutive queries in a query series have overlapping computations (e.g., when query series operate on a sliding window spanning multiple segments), these queries can be rewritten to expose the results of common intermediate computations (similar to materialized views in database systems) to be used by later queries in the same query series.

More importantly, with query series, query execution is now mostly driven by bulk updates to input streams rather than by submissions from users. Queries in different query series that operate on the same input stream can now be aligned and optimized together as one aggregated query. This helps remove redundancies, which are spatial correlations across query series. Given the power-law distribution on data stream accesses that we observe, a significant number of query series would access the most popular data streams and offer opportunities for optimizations when aligned and combined. To increase chances of sharing across query series, a query might be further 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 ensures that all queries on the same stream process the data on aligned segment windows, even if some queries originally process data over multiple segment windows.

3.1 Comet

We have developed Comet as a query processing engine that supports the BSP model and enables new optimizations for DISC systems. Comet allows users to submit a query series by specifying the period and the number of recurrences of the computations. We use the following terms to define the computation units in an 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: arrival of the segment with the maximum timestamp triggers execution of the SS-query. An S-query can be decomposed into one or more SS-queries in a 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, thereby leveraging any common I/O and computations among these SS-queries.

Figure 5 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. This allows Comet to align query series based on the segments they involve. As with current batch processing systems such as DryadLINQ, Comet carries out query optimizations with an emphasis on optimizing normalized jumbo-queries. Different from the flow in DryadLINQ, execution plans are not executed immediately. Instead, arrivals of new segments trigger executions of their corresponding jumbo-queries. Our implementation of Comet in DryadLINQ is presented in Section 4.

Comet collects statistics about data stream characteristics, operators, and custom functions for cost estimation. As done traditionally in database systems, cost estimation assesses tradeoffs involved in various optimization choices and chooses one with the lowest costs (Section 4.2); this capability is generally lacking in the current DISC systems. Collected statistics are stored in a *catalog* that is replicated on a set of machines for reliability.

Comet also allows users to submit ad hoc queries. Since ad hoc queries are executed on demand, Comet executes them with fewer optimizations and without normalization or combining them into the jumbo query. However, Comet does use statistics in the catalog for optimizing ad hoc queries if they involve the same data streams and the same custom functions as those in sbumitted query series.

4. INTEGRATION INTO DRYADLINQ

The BSP model can be introduced to existing DISC systems and leverage their software stacks. We have integrated Comet into DryadLINQ and implemented Comet optimizations.

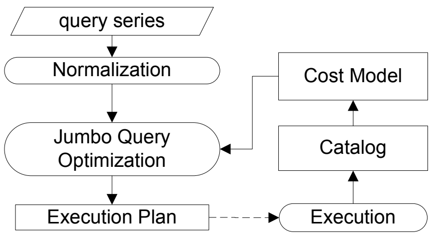


Figure 5: Comet execution flow for query series.

```
1  // Q1: daily custom grouping on (A,B,C)
2  q1 = env.Extractor("log?today")
3    .Select(x => new {x.A, x.B, x.C})
4    .Where(x => x.A != "gb")
5    .GroupBy(x => x) //grouping on (A,B,C)
6    .Select(x => new {x.Key, c = x.Agg()});
7
8  // Q2: weekly histogram aggregation grouping on (A,B)
9  q2 = env.Extractor("log?today-6...today")
10   .Select(x => new {x.A, x.B})
11   .Where(x => x.A != "gb")
12   .GroupBy(x => x) //grouping on (A,B)
13   .Select(x => new {x.Key, a = x.Count()});
14
15 // Q3: daily join on today and yesterday's segments
16 q3a = env.Extractor("log?today")
17   .Select(x => new {x.A, x.B, x.D})
18   .Where(x => x.A != "ru")
19   .GroupBy(x => x.D) //grouping on D
20   .Select(x => new {x.Key, m = x.Max(y => y.B)});
21 q3b = env.Extractor("log?today-1")
22   .Select(x => new {x.A, x.B, x.D})
23   .Where(x => x.A != "ru")
24   .GroupBy(x => x.D) //grouping on D
25   .Select(x => new {x.Key, m = x.Max(y => y.B)});
26 q3 = q3a.Join(q3b, x => x.m, y => y.m, (a, b) => a);
```

Figure 6: Three queries Q1, Q2, and Q3 in DryadLINQ. They resemble in structure the most popular query series in the trace. The input is a log generated from clicks in Microsoft Bing Search. Fields A–E are anonymous fields in the real log.

To describe our integration, we use three sample queries (Figure 6), 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. 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

Taking an ad hoc query as input, DryadLINQ processes it 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, with optimizations including pipelining and removal

[1]DryadLINQ also enables dynamic optimizations. We leave them out because they are not particularly relevant here.

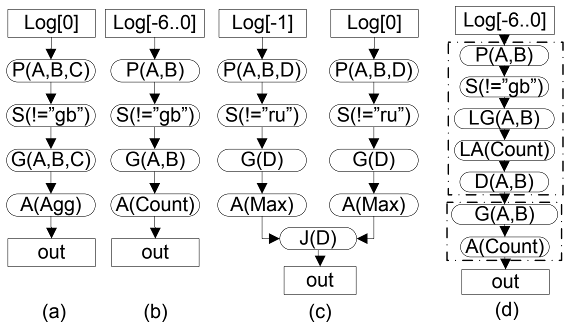


Figure 7: Logical plans for Q1, Q2, and Q3 ((a),(b), and (c)), and a physical plan for Q2 (d).

of unnecessary nodes. Each vertex in the Dryad execution graph has several physical operator nodes. The vertices are deployed to different machines for distributed executions.

Figure 7 shows the logical plans of the sample queries. In DryadLINQ, physical optimizations in Phase (2) take into account the underlying distributed system configurations and data properties. Figure 7 (d) shows the corresponding physical plan for the second query. During the transformation, DryadLINQ applies local reduction optimization: a local grouping (`LG(A,B)`) followed by a local aggregation (`LA(Count)`) on each machine. A distributed partition phase (`D(A,B)`) then shuffles locally grouped data at the cluster level for global grouping (`G(A,B)`) and aggregation (`A(Count)`).

The integration of Comet into DryadLINQ 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. Details on these integration issues are described in the following subsections.

4.2 Cost Model

A precise cost model is in general hard to attain, especially for DISC systems [19]. The integration of Comet alleviates this 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, while 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., input and output sizes for each operator, as well as cost of custom functions, and stores such information in the catalog for cost analysis.

Second, the integrated cost model focuses on estimation of total disk and network I/O. This is because 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 in the query, the current version of Comet avoids complications in cost models for traditional databases [25].

As a result, we have implemented a simple and effective cost model. At each stage, Comet can take the input size of a query and use the relationship between input and output sizes from a previous run of the same execution to estimate the amount of I/O. We estimate the output size

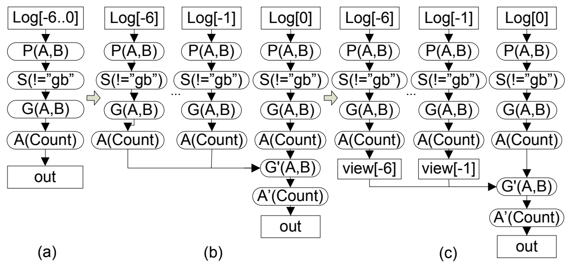


Figure 8: S-query normalization on Query Q2. (a) original logical plan for `Q2`, (b) after decomposition, (c) after adding materialized views.

of a stage based on the selectivity of filters and the ratio of input/output sizes from a previous run of the same execution. Our experiments have validated the accuracy of such a simplified cost model, and its effectiveness in guiding optimization choices (Section 5).

We also add a step of consulting the cost model and the catalog into Phase (3) of DryadLINQ execution, especially when the benefit and cost of some optimizations are dependent on certain properties of queries and their input data streams. To allow for an iterative optimization process, rather than following the pipelined process in DryadLINQ, we add a control loop between Phases (2) and (3) of DryadLINQ execution, 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. Cost estimation in Comet significantly reduces the needs for such optimizations.

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 of which is triggered when a segment of an input stream becomes available. This process essentially turns an S-query into an incremental computation (see [23]). In the worst case, the normalization will convert an entire S-query into a single SS-query.

Figure 8 depicts the normalization process for an S-query of the second sample query series. As the S-query involves one week's data (Figure 8 (a)), we split the input node into seven nodes, each corresponding to one daily segment (Figure 8 (b)). Comet then explores the paths starting from those nodes, examines each operator at each level from the top down, and splits the operator whenever it is appropriate based on the 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 8 (c)). There are also others, such as aggregations with some custom functions, that cannot be decomposed because we cannot easily make a decomposed plan which produces 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 the 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 however has a different structure and requires careful construction.

We arrive at Figure 8 (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 SS-qeuries with the same timestamp, as shown in Figure 9 (a), for further optimizations. Through normalization, redundancies across queries are exposed to later logical and physical optimizations of Comet-enabled DryadLINQ.

4.4 Logical Optimization

Comet enables new logical optimizations including shared computations and reused views to remove redundancies in the logical representation of jumbo queries. These techniques are rooted in logical query optimizations in database systems [24, 31].

Shared computations. While current DryadLINQ can identify shared computations 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.

Operator reordering generates multiple possible plans, and the cost model evaluates which one is better. Consider two branches that have two different selection operators followed by two grouping operators with the same grouping keys. We can swap the selection and grouping operators, so as to do the grouping operation only once. However, if the two selections can reduce input data size dramatically, this optimization actually hurts overall performance. Note that current DryadLINQ chooses the latter case. Our evaluation shows that this choice is not always true: a wrong decision may result in I/O penalty.

Reused view across jumbo-queries. Redundant computations can also occur across jumbo queries, which happens when two jumbo queries operate on overlapping input segments. For instance, the output of `A(Max)` in Figure 9 (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 a *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 [18, 19], co-location of the results across jumbo queries needs special care on data replication. Comet replicates a partitioned view according to the partitioning scheme of related views in the catalog.

One example of partitioned views with co-location is shown

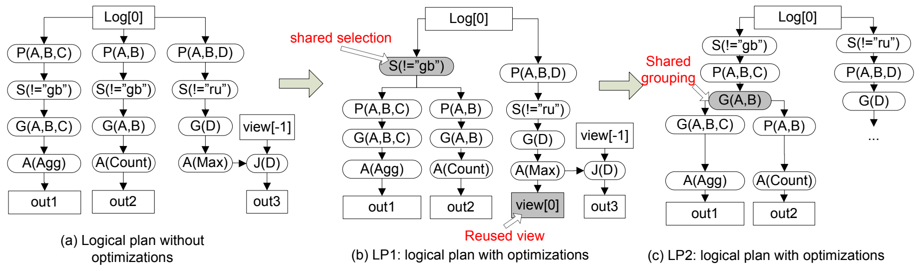


Figure 9: Logical optimizations of Comet, generating two logical-plan candidates: LP1 and LP2.

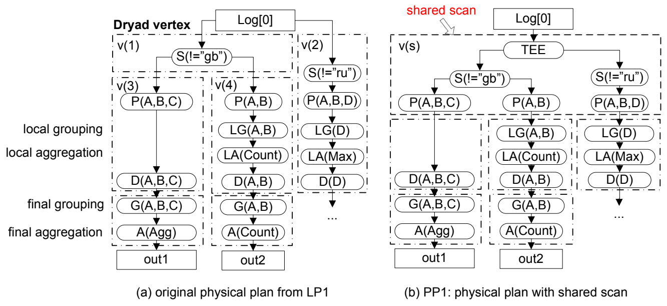


Figure 10: An example of shared scan: original physical plan directly generated from LP1 and the physical plan PP1 with the shared scan optimization.

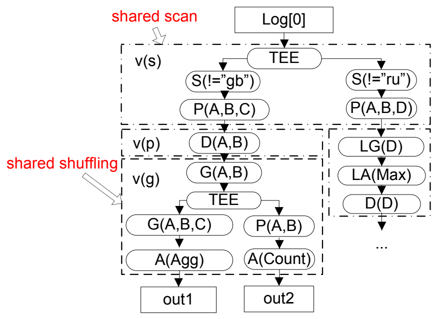


Figure 11: An example of shared shuffling: the physical plan generated from LP2.

in Figure 8 (c), where the final SS-query merges the results from all the previous SS-queries. Comet partitions `view[-6]`, ..., `view[-1]` in the same way and co-locates the corresponding partitions in the same machine, so that aggregation can be done locally on each partition.

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 9 (b).) There are two types of branches, one enables *shared scan* [1] and the other *shared shuffling*. The differences lie in whether a branching point corresponds to local data transfers or network data transfers.

Shared scan. An example of shared scan is shown in Figure 10 (a): `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 10 (a)), 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 connecting 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 10 (b)), so that they can cooperate with each other to reduce the number of scans on the input segment to only one.

Whether or how to package branching nodes into a vertex is a decision to be made based on the cost model. Since the current design of DryadLINQ outputs the result of a Dryad vertex to disk, putting multiple branching nodes in a Dryad vertex causes the results of all the branching nodes written to disk, rather than pipelining to next processing steps. Therefore, naively putting all branching nodes into a vertex is not always beneficial. The cost model decides whether putting a node into a shared-scan vertex according to the relative sizes of input and output.

Shared shuffling. An example of shared shuffling can be found in Figure 9 (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 9 (c)). Then, it transforms this logical plan to a physical one (Figure 11), 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 further operators for the two SS-queries into one vertex (`v(g)`).

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 redundancies across queries. More concretely, two different physical execution plans for the same jumbo query are shown in Figure 10 (b) and Figure 11. 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-shuffling optimization to reduce redundant data shuffling across SS-queries. Unlike DryadLINQ, which always chooses the former, Comet relies on the cost model to predict which one is better.

5. EXPERIMENTS

In this section, we evaluate Comet with DryadLINQ on a small cluster and with simulations on the real trace.

5.1 Experimental Setup

We perform two sets of experiments to evaluate Comet. The first set of experiments is on a real deployment of Comet on a 40-machine cluster using a micro benchmark. This micro benchmark is to reveal micro-level details of Comet with full control of choices on Comet optimizations in DryadLINQ. The second set of experiments performs simulations on the entire real-world trace reported in Section 2 to assess global effectiveness of Comet.

We mainly focus on system throughput, and use *total I/O* as the main metric. Total I/O is the number of bytes (in GB) in both disk and network I/O during an execution.

To evaluate the separate benefits of individual optimization techniques, we manually enable/disable certain optimizations in both real deployment and simulations. 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.

5.1.1 Micro benchmark setup

All 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.

We assemble a micro benchmark, which consists of the three query series in Figure 6. The three sample queries resemble the key structures of the most popular query series in the trace and with different characteristics in different dimensions: 1) periods: `Q1` and `Q3` are daily queries, and `Q2` consists of weekly queries; 2) complexity: `Q1` and `Q2` are simple queries with only one input stream, and `Q3` has a join on two inputs; 3) grouping: `Q1` and `Q2` have a common grouping prefix (A, B), and `Q3` has a grouping on D. These characteristics are to evaluate the key optimization techniques in Comet.

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 referenced columns (denoted as A–E) into a stream. Each segment is evenly partitioned and stored on machines in a cluster. The average size of each segment is around 16 GB. The total stream size is around 112 GB in total, covering updates for a week. 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 optimal execution plans for jumbo queries are automatically generated according to the cost model in Comet. In the experiment, we also examine execution plans without Comet optimizations. 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 execution plan, *Logical* uses the plan in Figure 10 (a), and *Full* uses the plan in Figure 11 (we will discuss why Comet did not select Figure 10 (b) later.) For both *Logical* and *Full* on the input of one week, 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.

We have run each experiment five times. Variances among different runs are small, and we report averages.

5.1.2 Simulation setup

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: ad hoc queries and recurring queries. 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. If statistics of an input segment are available, we estimate the total I/O cost of an SS-query with our cost model. Otherwise, we evenly distribute the total I/O costs of each step of the original query to those of 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 shuffling: the cost of only one input scan or partitioning is counted.

The Comet simulator also supports simulating executions of ad hoc queries. Due to the unpredictability of ad hoc queries, the simulator considers whether an ad hoc query can reuse views generated from previous executions of query series. Finally, the Comet simulator outputs the total I/O cost of the simulated workload at the end of simulation.

In this study, we use the trace described in Section 2 and report the simulation results. This offers the capability of estimating the effectiveness of Comet with a real trace from a production system.

5.2 Micro Benchmark Study

The micro benchmark study evaluates the cost model and choices in jumbo-query optimizations, including shared scan and shared shuffling.

Cost Model Accuracy. We evaluate the accuracy of our cost model by comparing total I/O numbers from real runs and from estimation based on statistics from previous runs; that is, we use statistics from previous execution and the input size of the current one to estimate the cost of the current execution. The experiment was done twice under

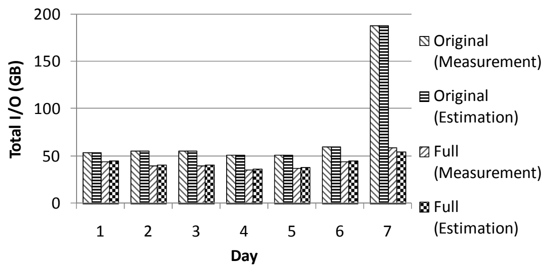


Figure 12: Estimated and measured total I/O from day 1 to day 7.

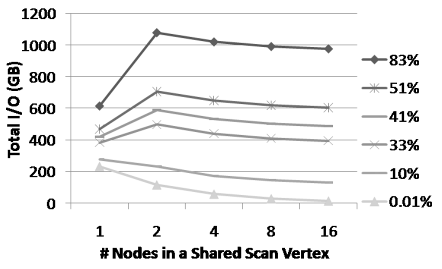


Figure 13: Total I/O of the shared scan.

the *Original* and *Full* configurations. Figure 12 shows the result: the estimated total I/O follows the actual total I/O closely, which validates the accuracy of our cost model.

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 13 shows the total I/O with different selectivities (by changing filtering conditions). The results show that it is beneficial to enable shared scan with a large number of branches when selectivity is low: this is the case for most queries in our trace. When selectivity is high, the overhead of materializing results in shared scan is high, and the appropriate number of queries per shared scan vertex is small. Our cost model can guide this decision making.

Early filtering versus shared shuffling. As discussed in Section 4.5, pushing down a selection operator for shared shuffling is not always profitable. We investigate how total I/O varies with selectivity of the selection operators in the first query series (by changing filtering conditions.) The experiment was done twice on two instances of the first query series: the first instance applies early filtering without shared shuffling; the second instance applies late filtering with shared shuffling. Figure 14 shows the total I/O numbers for the two cases with varying selectivity: As selectivity increases, the benefit of shared shuffling increases. When the selectivity is larger than 50%, late filtering is preferable. Comet decides early filtering or shared shuffling using cost estimation according to selectivity.

Early aggregation versus shared shuffling. Recall that Comet selected Figure 11 (w/ shared shuffling) instead of Figure 10 (b) (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

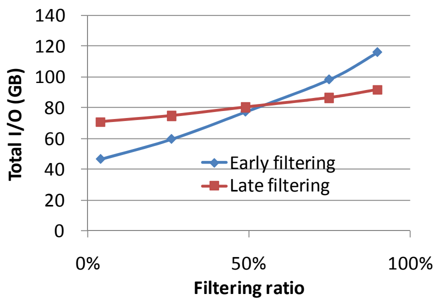


Figure 14: Performance on running two instances of `Q1`.

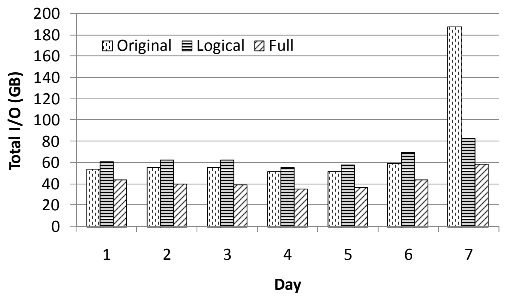


Figure 15: The effectiveness of Comet on the micro benchmark.

0.2 GB of the network I/O, due to the distribution of partitioning key `A,B` for the aggregation. Meanwhile, shared shuffling 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 with and without co-locating 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%.

To put it all together, we run the three query series under the three optimization configurations. Figure 15 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.

With logical optimization, *Logical* and *Full* reduce the total I/O by 12.3% and 42.3%, respectively. Besides, since our optimization divides the execution of Q3 into seven days, *Logical* and *Full* have a more balanced load compared to *Original*.

5.3 Global Effectiveness Study

Simulation validation. We first evaluate the accuracy of our simulation. Figure 16 shows the total I/O of the micro benchmark reported in the experiments described earlier and the result with our simulation, under the three optimization configurations. 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 two optimization configurations. By default, the optimization configurations have query normalization enabled. To study effectiveness of normalization, we also

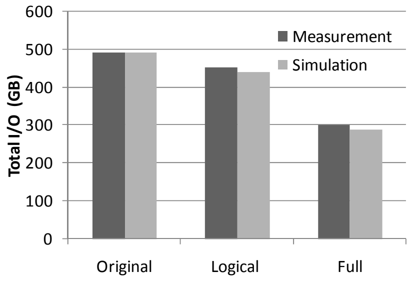


Figure 16: Simulation validation on micro benchmark.

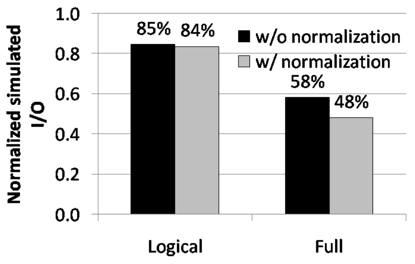


Figure 17: Total I/O in simulation with and without query normalization.

run simulation without normalization. Figure 17 depicts the simulated results under these four configurations. It shows that the *Full* optimization with normalization can reduce total I/O by 52%, a significant cost saving. The *Full* optimization without normalization can also reduce total I/O by 42%. This means that normalization contributes 10% of cost saving; this is reasonable given 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% reduction in terms of total I/O.

Performance gain breakdown. A closer look at savings from logical optimizations reveals that around 76% and 22% of their savings come from extraction and aggregation operations, with around 2% due to the shared computations 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 shuffling.

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

Load distribution. Figure 18 shows the amount of per-day I/O of *Original* and *Full* optimization normalized to that of maximum of *Original* in a month. We observed that our optimization techniques reduce total I/O for every day in the month. The normalized daily load of *Original* is between 0.03 and 1.00, and that of *Full* is between 0.02 and 0.65. With query decomposition, load in the system with *Full* becomes more balanced than that with *Original*.

Comparison with existing approaches. Finally, we implemented two existing complementary multi-query optimization approaches in the simulator, one based on materialized views [19] for caching (denoted as *MV*), and the other scheduling input scans [1] (denoted as *SM*). *MV* identifies common computations based on the past history and stores the results of identified common computations for reuse when executing in the current day. *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.

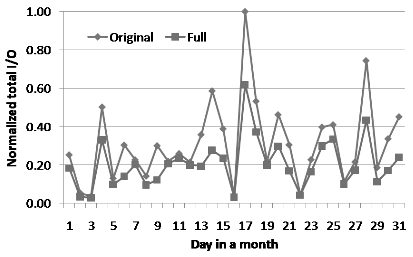


Figure 18: Normalized daily I/O with full optimization.

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%

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* to the system. Comet is more effective than *MV+SM*, with 36% less I/O. The improvement is mainly due to two factors. First, Comet performs query decomposition for sharing whereas neither *MV* nor *SM* does. Second, Comet has a larger optimization scope than *MV+SM*. Comet aligns query executions to data updates for optimizations among query series, whereas *MV+SM* bases its scope on query arrivals.

6. RELATED WORK

Comet builds on prior work in both data intensive distributed systems and databases systems.

6.1 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 [10], Dryad [18], and Hadoop [16] provide scalable and fault-tolerant execution of individual data analysis tasks. More recently, high-level languages, such as Sawzall [21], Pig Latin [20], DryadLINQ [30], and SCOPE [6] introduce high-level programming languages, often with certain SQL flavors, to facilitate specification of intended computation. All these systems adopt a batch 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.

I/O optimizations for DISC systems have made query optimization techniques in database systems relevant. Olston et al. [19] recognize the relevance of database optimization techniques and propose a rule-based approach for both single- and multi-query optimizations in batch processing. Existing systems such as DryadLINQ [30] and SCOPE [6] have already employed database techniques to improve system performance. Chen et al. [8] optimize data distribution in evaluating composite aggregation queries. Agrawal et al. [1] propose shared scans of large data files to improve I/O performance. 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.

Our previous work [17] presents a preliminary study on the trace and outlines some research opportunities. This work extends the previous study with an emphasis on query correlations, and realizes the research opportunities through the proposal of the BSP model and the integration into DryadLINQ.

6.2 Database Optimizations

Many core ideas in Comet optimizations can find their origins in query processing techniques in database systems [11, 15], both in batch processing [11, 24, 26] 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 computations are 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 [11]. Shared-nothing database systems like Gamma [12] and Bubba [5] focus mainly on parallelizing a single query. As for multiple query optimizations, materialized views [2, 24] 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 [31]. In Comet, persistent outputs registered in a catalog are materialized views, without complicated and usually expensive view maintenance in database systems [4]. In addition, by merging queries into a jumbo query, results of most common expressions do not need to be materialized. Concurrent disk scans on the same relational table can be shared with proper scheduling [9, 32, 29].

Stream processing. Stream processing systems such as STREAM [28] 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 [27] and for identifying common sub-queries among continuous queries [7] are proposed to process streams efficiently.

7. 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 consist 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 an Underlying System. Comet can also benefit from better support from 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 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 shuffling, multiple queries are now scheduled to run on the same machines concurrently. While this optimizes overall throughput and improves resource utilizations, it might create hotspots in some part of the system, with idle resources elsewhere. A distributed storage system must balance data allocation; this will help alleviate tensions 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 [13]. 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. We believe that a combination of automatic program analysis and tasteful constraints on the custom functions might help address the issues.

8. CONCLUDING REMARKS

With an increasing use of distributed systems in large-scale data processing, we envision the inevitable convergence of database systems and distributed systems in this context. The convergence will bring a set of new challenges and opportunities in performance optimization. Comet is a step towards that convergence. Motivated by the observations from a real system, Comet embraces a new execution model driven by arrivals of data updates and makes cross-query optimization tractable. While Comet leverages DryadLINQ, an existing system, for system-level resource management and scheduling, the underlying distributed system challenges have not yet been addressed satisfactorily, especially considering a large-scale and dynamic system with a large number of incoming and concurrent queries.

Acknowledgement

We would like to thank the Bing Search team for access to the production trace. We are also grateful to David Chu, Sean McDirmid, Geoffrey Voelker, Yuan Yu, Jingren Zhou, Li Zhuang, the System Research Group of Microsoft Research Asia, as well as the anonymous reviewers, for their insightful comments.

9. 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] L. Chen, C. Olston, and R. Ramakrishnan. Parallel evaluation of composite aggregate queries. In *ICDE*, 2008.

[9] L. S. Colby, R. L. Cole, E. Haslam, N. Jazayeri, G. Johnson, W. J. McKenna, L. Schumacher, and D. Wilhite. Redbrick Vista: Aggregate computation and management. In *ICDE*, 1998.

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

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

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

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

[25] 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.

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

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

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

[29] X. Wang, R. Burns, and T. Malik. Liferaft: Data-driven, batch processing for the exploration of scientific databases. In *CIDR*, 2009.

[30] 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.

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

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