UNIVERSITY OF CALIFORNIA
RIVERSIDE

Revisiting Aggregation Techniques for Data Intensive Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jian Wen

December 2013

Dissertation Committee:

Dr. Vassilis J. Tsotras , Chairperson
Dr. Vagelis Hristidis
Dr. Eamonn Keogh
Dr. Srikanth Krishnamurthy

The Dissertation of Jian Wen is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I would like to express my gratitude to my family. Mom, Dad, and Jingjing, this thesis is dedicated to you, for your sacrifices and for the worries you have had on me day and night. You are my source of power.

I want to express my deepest gratitude to my advisor, Professor Vassilis Tsotras, for providing me the continuous guidance, support and encouragement all the way along my doctoral study, and for being a mentor guiding me through the obstacles and difficulties in the research with the greatest patience, and for being a friend always ready to help no matter when I ask, and for giving me the maximum freedom for my research, and for sharing me his experiences and insights to broaden my horizon, and, the most importantly, for showing me the meaning of family.

Thanks to Dr. Donghui Zhang, for opening the door and leading me into the world of database research, and for teaching me the value of introspection and self-confidence.

Thanks to Professor Michael Carey at UC Irvine, for always being available to advise and guide me, and for teaching me how to think as a top database researcher, and for the great experience in working with the AsterixDB project.

I would also like to thank Vinayak Borkar, for guiding me to be both an engineer and a theorist, and for showing me a great system architect's point of view. He has been a "always-online" support through skype for any of my questions in both research and development.

Thanks to my PhD committee, Dr. Vagelis Hristidis, Dr Eamonn Keogh and Dr. Srikanth Krishnamurthy, for the great support and valuable comments for my research.

Thanks to my UCR DB Lab fellows, namely: Mahbub Hasan, Marcos Vieira, Wenyu Huo, Michael Rice, Eldon Carman, Steven Jacobs, Petko Bakalov, Theodoros Lappas, for the thoughtful discussions and the happy lab time. I am also grateful to the UCI AsterixDB team: Professor Chen Li, Yingyi Bu, Alexander Behm, Raman Grover, Pouria Pirzadeh, Zachary Heilbron, Young-Seok Kim, Keran Ouaknine. It has been a great teamwork with many insightful meetings and fun debugging time!

Finally, I express my love and gratitude to my dear Winnie, for always being there, for me and with me.

To my dear Weilin, Mom, Dad and Jingjing.

ABSTRACT OF THE DISSERTATION

Revisiting Aggregation Techniques for Data Intensive Applications

by

Jian Wen

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2013
Dr. Vassilis J. Tsotras , Chairperson

Aggregation has been an important operation since the early days of relational databases. Today's Big Data applications bring further challenges when processing aggregation queries, demanding robust aggregation algorithms that can process large volumes of data efficiently in a distributed, share-nothing architecture. Moreover, aggregation on each node runs under a potentially limited memory budget (especially in multiuser settings). Despite its importance, the design and evaluation of aggregation algorithms has not received the same attention that other basic operators, such as joins, have received in the literature.

This dissertation revisits the engineering of efficient aggregation algorithms for use in Big Data platforms, considering both local and global aggregations. We firstly discuss the salient implementation details and precise cost models of several candidate local aggregation algorithms and present an in-depth experimental performance study to guide future Big Data engine developers. We show that the efficient implementation of the local aggregation operator for a Big Data platform is non-trivial and that many factors, including memory usage, spilling strategy, and I/O and CPU cost, should be considered. Then we show extended cost models that can precisely depict the cost of

global aggregation plans, considering not only the local cost factors but also the network cost. We discuss a generic framework to describe a tree-structured global aggregation plan, and propose a cost-model based algorithm for efficiently finding the non-dominated global aggregation plans for different output properties, given the input data statistics and the global computation resources.

Furthermore, spatial and temporal information introduces more semantics to traditional aggregations, requiring specific efficient algorithms that could utilize the additional spatial and temporal information during the query processing. In the last part of the thesis we show a novel aggregation application for monitoring the top-k unsafe moving objects in a continuous data stream where the spatial and temporal information change. We show that such a query can be generalized as an extended aggregation operation where the grouping condition is unknown without looking at all valid data in the given query time window. We then propose I/O-efficient algorithms to answer such queries utilizing spatial and temporal index structures.

Contents

List of Figures

List of Tables

1

Introduction

"Objects are grouped by their categories."

– Xi Ci I

1.1 Aggregation

Aggregation has always been a very important operation in database processing. For example, all 22 queries in the TPC-H Benchmark [3] contain aggregation. It is also a key operation for data preprocessing and query processing in data intensive applications, such as machine learning on large volume data [12], and web-related data processing like web-log and page ranking [44], etc.

Aggregation operations in such data processing tasks usually can be divided into to categories, both of which have been well-defined in the academic literature [17] and also the industry standard like SQL. The first category is called **scalar aggregation**, where a single aggregation value is returned by aggregating the input data. The other category is called **aggregate functions**, which firstly groups the input data based on the given grouping keys, and then applies the aggregation functions for all the records

of each group. Due to the nature of applying group-wise aggregations, this category is also known as **group-by** operation.

To illustrate these two categories in SQL, here we consider the "big data" dataset **UserVisits** from [44]; it contains a visit history of web pages with the attributes shown in Table 2.1.

Attribute Name	Description
`sourceIP`	the IP address (the source of the visit)
`destURL`	the URL visited
`adRevenue`	the revenue generated by the visit
`userAgent`	the web client the user used
`countryCode`	the country the visit is from
`languageCode`	the language for the visit
`searchWord`	the search keyword
`duration`	the duration of the visit

Table 1.1: Attributes in UserVisits dataset.

One example for the scalar aggregation is to get the total advertisement revenue and also the total number of unique source IP addresses from this dataset. The SQL query can be written as:

```
SELECT sourceIP, SUM(adRevenue), COUNT(DISTINCT sourceIP)
FROM UserVisits
```

For this query both aggregation operations (`SUM` and `COUNT`) are applied directly to the whole column of interest. Note that we can also use the keyword `DISTINCT` to eliminate the duplicates before applying the aggregation operation.

An example of an aggregation function query appears in the following SQL query, which for each `sourceIP` address (representing a unique user), computes the total advertisement revenue and the total number of visits:

```
SELECT sourceIP, SUM(adRevenue), COUNT(*)
FROM UserVisits
GROUP BY sourceIP
```

The main difference here is that this query contains a `GROUP BY` clause to indicate the grouping column. Multiple columns can be used in the `GROUP BY` clause. Note that as far as the grouping columns are the same, different aggregation functions can be processed together in a single aggregation procedure. Multiple aggregation functions over the same group-by columns can be considered as a single "super" aggregation function containing all these aggregation functions as internal states. Scalar aggregations can be considered as the special case of the aggregation functions where the whole input dataset is in a single group. In this study we focus on the aggregation functions, so without explicitly specification, we will use "aggregations" and "aggregation functions" interchangeably through this writeup.

1.2 Motivation and Challenges

For aggregation in big data applications, the input data is always spread over a distributed environment, like Hadoop and many popular distributed relational databases. Aggregation is typically processed in a map-combine-reduce fashion. Such a strategy first obtains the local aggregation results ("map" and "combine" phase), which are then merged to get the global aggregation results ("reduce" phase). The physical implementation of such a map-combine-reduce procedure for aggregations could be varied a lot due to the different hardware environments. Different hardware environments have different cost factors for the primitive operations the aggregation will use, like comparison, hashing, data loading and storing etc. Different aggregation algorithms always

have significant different usage patterns on these primitive operations.

Given a certain hardware environment, an interesting (and also not trivial) question is: how to find a cost-efficient implementation of an aggregation task for a specific hardware environment. The "cost" could be evaluated through different measurements during the experiments, like the total elapse time or the total computation resource consumption during the whole aggregation operation.

To pick an efficient global aggregation plan, both local and global aggregation strategies should be considered. The local strategy is about the aggregation algorithm to be used to process a single partition of the input data, and the global strategy is about transferring and redistributing the results of local aggregations to get the final aggregation result.

In this thesis, I discuss a cost model based study for both local and global aggregation plans. Chapter 2 will focus on the local aggregation, including the algorithms and implementation details for several well-known local aggregation algorithms. The most important contribution for this local aggregation work is that we propose a very precise cost model to all the algorithms listed in that chapter. We also did extensive experiments to verify the correctness of our model. Since our model is built over some basic components for sorting and hashing, the model can be extended for other similar algorithms utilizing the sort and hash strategies.

Then in Chapter 3, I show the extended work on the global aggregation plans in a shared-nothing environment. In this chapter we describe the general physical aggregation structure in such a environment, called aggregation tree. Then we will show the different components affecting the overall performance of the global aggregation plans, including the network structure, local aggregation algorithms and data redistribution connectors. For the local aggregation algorithms, we also extended our local aggrega-

tion study to address the partial aggregation algorithms that can be used in a global plan. To answer the question on efficient global aggregation plans, we extended our local cost model to predicate the global aggregation cost factors in a hardware-independent way. We then proposed a plan searching algorithm to find the non-dominated global aggregation plans without scanning the whole plan space.

Finally in Chapter 4, we discuss a novel aggregation application for spatial and temporal data. This new aggregation application is different from the traditional aggregation work covered in Chapter 2 and 3 in the sense that spatial and temporal information should be considered when identifying the groups for aggregation. Furthermore, we claim that this novel aggregation work should be processed to utilize the characteristics of the spatial and temporal data. In this chapter we show this problem in a specific real scenario for public safety enforcement, and propose our efficient algorithms utilizing index structures.

2

Local Aggregation

In this chapter we discuss the implementations and also cost models for several local aggregation algorithms. Although several of them have been well adopted by different systems, in our effort to support the aggregation operation in our next generation parallel data processing platform AsterixDB [6], we found that to correctly implement them in order to guarantee the efficiency and skew tolerance, is not a trivial work. Specifically we noticed two new challenges that big data applications impose on local aggregation algorithms: first, if the input data is huge and the aggregation is group-based (like the "group-by" in SQL, where each unique group will have a record in the result set), the aggregation result may not fit in main memory; second, in order to allow multiple operations being processed simultaneously, an aggregation operation should work within a strict memory budget provided by the platform.

Furthermore, for these well-known algorithms, like pre-sorting the input data [17], or using hashing [50], have not been fully studied with respect to their performance for very large datasets or datasets with different distribution properties. While some join processing techniques [23] can be adapted for aggregation queries, they are tuned

for better join performance. All these existing algorithms lack for details on how to implement them using strictly bounded memory, and there is no study about which aggregation algorithm works better for which circumstances. To answer these questions we present in this chapter a thorough study of single machine aggregation algorithms under the bounded memory and big data assumptions. Our contributions can be summarized as:

1. We present detailed implementation strategies for six aggregation algorithms: two are novel and four are based on extending existing algorithms. All algorithms work within a strictly bounded memory budget, and they can easily adapt between in-memory and external processing.

2. We devise precise theoretical cost models for the algorithms' CPU and I/O behaviors. Based on input parameters, such models can be used by a query optimizer to choose the right aggregation strategy.

3. We deploy all algorithms as operators on the Hyracks platform [8], a flexible, extensible, partitioned-parallel platform for data-intensive computing, and evaluate their performance through extensive experimentation.

The rest of this chapter is organized as follows: Section 2.1 presents related research, while Section 2.2 discusses the processing environment for our aggregation algorithms. Section 2.3 describes in detail all algorithms and Section 2.4 presents their theoretical performance analysis. The experimental evaluation results appear in Section 2.5. Finally Section 2.6 summarizes this chapter. In the Appendix we list the theoretical details of the basic component models used in our cost model analysis in Section 2.4.

2.1 Related Work

In our search for efficient local aggregation algorithms for AsterixDB, we noticed that aggregation has not drawn much attention in the study of efficient algorithms using tightly bounded memory. The well-known sort-based and hash-based aggregation algorithms discussed in [17], [7], [45] and [52] provide straight-forward approaches to handle both in-memory and external aggregations, but these algorithms use sorting and hashing in a straight-forward way and there is space to further optimize the CPU and I/O cost.

[21] discussed three approaches for aggregations that may not fit into memory, namely nested-loop, sort-based and hash-based. It suggests that the hash-based approach using hybrid-hash would be the choice when the input data can be greatly collapsed through aggregation. Our study of the hybrid-hash algorithm reveals that its hashing and partitioning strategy can be implemented in different ways, leading to different performance behaviors. These have not been discussed in the original paper, and precise cost models are also missing for the proper selection of aggregation algorithms under different configurations. [23] presented optimizations for hybrid-hash-based algorithms, including dynamic destaging, partition tuning and many best-practice experiences from the experience of SQL Server implementation. However, this chapter focuses more on optimization related to joins rather than aggregations. [50] tried to address the problem of efficient parallel aggregation algorithms albeit for SQL, as we are doing for the AsterixDB project. For the local aggregation algorithm, they picked a variant of the hybrid-hash aggregation algorithm that shares its hashing space among all partitions. But no optimization has been done with other aggregation algorithms.

More recently, [13] examined thread-level parallelism and proposed an adaptive

aggregation algorithm optimized for cache locality by sampling and sharing the hash table in cache. However, in order to reveal the performance benefits from using the CPU cache, only in-memory aggregation algorithms were addressed. We think that for an external aggregation algorithm, it is important to address the I/O efficiency first, and then to optimize the CPU behavior for each in-memory run of the aggregation. [58] studied several in-memory aggregation algorithms for efficient thread-level parallelism and reducing cache contention. Similar to our proposed Pre-Partitioning algorithm, the PLAT algorithm in their paper partitions the input data based on their input order and fills up the per-thread hash table first. However, PLAT processes records in memory even after the hash table is full, based on the assumption that the input data can be fit into memory. In our algorithm we explore the case where the memory is not enough for in-memory aggregation, so disk spilling happens after the hash table is full. In our experiments we also observe significant hash miss cost in our Pre-Partitioning algorithm, and we use an optimized hash table design to solve this problem.

2.2 Processing Environment

We now proceed to describe the main characteristics of the aggregation operation that we consider as well as the assumptions about the data and resources used.

2.2.1 Aggregate Functions

Our focus is on aggregate functions [17] such as aggregation combined with the "GROUP-BY" clause in SQL. As an example, consider the "big data" dataset **UserVisits** from [44]; it contains a visit history of web pages with the attributes shown in Table 2.1.

Attribute Name	Description
`sourceIP`	the IP address (the source of the visit)
`destURL`	the URL visited
`adRevenue`	the revenue generated by the visit
`userAgent`	the web client the user used
`countryCode`	the country the visit is from
`languageCode`	the language for the visit
`searchWord`	the search keyword
`duration`	the duration of the visit

Table 2.1: Attributes in UserVisits dataset.

An example `GROUP BY` aggregation appears in the following SQL query, which for each `sourceIP` address (representing a unique user), computes the total advertisement revenue and the total number of visits:

```
SELECT sourceIP, SUM(adRevenue), COUNT(*)
FROM UserVisits
GROUP BY sourceIP
```

The **by-list** (the `GROUP BY` clause in the example) specifies the **grouping key**, while the **aggregate function(s)** (`SUM` and `COUNT` in the example) specify the way to compute the **grouping state**. The grouping state in the above example has two aggregated values (sum and count). The result of the aggregation (the **group record** or **group** for short) contains both the grouping key and the grouping state.

Many commonly-used aggregate functions, like the `SUM` and `COUNT` in the example, can be processed in an accumulating way, i.e., for each group, only a single grouping state (with one or more aggregate values) needs to be maintained in memory, no matter how many records belong to this group. Similar **bounded-state** aggregate functions include `AVERAGE`, `MIN`, and `MAX`. Many other aggregate functions, like finding the longest string of a given field per group (i.e., "find the longest `searchWord` for each `sourceIP` in the `UserVisits` dataset"), can be considered as bounded-state functions if the memory

usage of the grouping state is bounded (for example the `searchWord` could be at most 255 characters long, which is a common constraint in relational databases). A query with multiple aggregate functions on the same group-by condition is also bounded on the state, as far as each of them is a bounded-state function. So all our discussion in this chapter also applies to this case.

However, there are aggregate functions that are not in the bounded-state category. An example is `LISTIFY` (supported in AsterixDB) which for each group returns all records belonging to that group in the form of a (nested) list. Since the size of the grouping state depends on the number of group members, its memory usage could be unbounded. In this chapter we concentrate primarily on bounded-state aggregate functions, as those are the most common in practice. Note that the simpler, scalar aggregation can be considered as an aggregate function with a single group (and thus all algorithms we will discuss can be applied to scalar aggregation directly).

2.2.2 Data and Resource Characteristics

We assume that the size of the dataset can be far larger than the available memory capacity, so full in-memory aggregation could be infeasible. Whether our algorithms use in-memory or external processing depends on the total size of the grouping state, which is decided by the number of unique groups in the dataset (grouping key cardinality), and also the size of the grouping state. An efficient aggregation algorithm should be able to apply in-memory processing if all unique groups can fit into memory, and shift dynamically to external processing otherwise.

This thesis assumes a commonly-used frame-based memory management strategy, which has been implemented in the Hyracks [8] data processing engine where all our algorithms are implemented. The Hyracks engine manages the overall system mem-

ory by assigning a tightly bounded memory budget to each query, in order to support parallel query processing. We will use M to denote the memory budget (in frames or memory pages) for a particular aggregation query, R to denote the size of the input data in frames, and G the size of the result set in frames.

For aggregation algorithms that utilize a hash table, current Hyracks operators use a traditional separate chaining hash table with linked lists [35]. The memory assigned to a hash table is used by a **slot table** and a **list storage area**. The slot table contains a fixed number of slots H (i.e., it is static hashing; H is also referred to as the **slot table size**). Each non-empty slot stores a pointer to a linked list of group records (whose keys were hashed to that slot). The list storage area stores the actual group records in these linked list(s). Group records from different slots can be stored in the same frame. A new group is hashed into a slot by being inserted to the head of the linked list of that slot (or creating a new linked list if the slot was empty). An already-seen group is aggregated by updating its group record in the linked list.

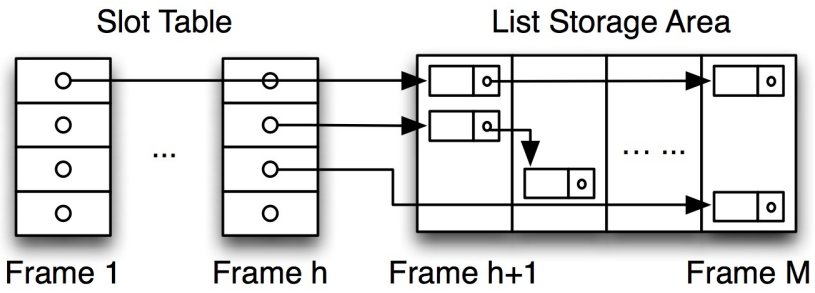


Figure 2.1: An In-memory Hash Table.

An in-memory hash table is **full** when no new group record can fit in the list storage area based on its given memory budget. Figure 2.1 shows such an in-memory hash table with a budget of M frames (for both the slot table and the list storage area), where h frames are occupied by the slot table.

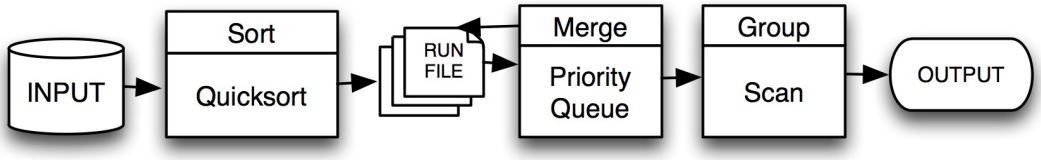


Figure 2.2: Sort-based Algorithm

2.3 Aggregation Algorithms

This section takes an in-depth look at six candidate aggregation algorithms: the Sort-based, the Hash-Sort, and four hybrid-hash-based algorithms (Original Hybrid-Hash, Shared Hashing, Dynamic Destaging, and Pre-Partitioning). The Hash-Sort and Pre-Partitioning algorithms are novel, while the others are based on adapting approaches discussed in the previous literature. Table 2.2 gives an overview of these algorithms.

2.3.1 Sort-based Algorithm

The classic Sort-based aggregation algorithm includes two phases, **sort** and **aggregate**. Figure 2.2 depicts the algorithm's workflow. The sort phase sorts the data on the grouping key (using a sort-merge approach), while the aggregate phase scans the sorted data once to produce the aggregation result. In detail:

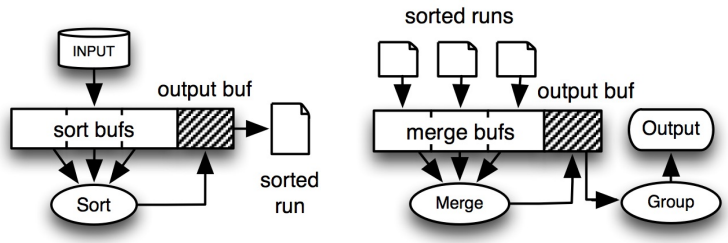
- Phase 1 (External Sort): (i) **Sort**: Data is fetched into memory in frames. When the memory is full, all in-memory records are sorted using the Quicksort algorithm [47], and flushed into a run file. If the total projected input data size is less than the memory size, the sorted records are maintained in memory and no run is generated. Otherwise, runs are created until all input records have been processed. (ii) **Merge**: Sorted runs are scanned in memory, with each run having one frame as its loading buffer. Records are merged using a loser-tree [35]. If the number of

Algorithm	Using Sort?	Using Hash?
Sort-based [17],[7],[45], [52]	Yes	No
Hash-Sort (New)	Yes	Yes
Original Hybrid-Hash [49]	No	Yes
Shared Hashing [50]	No	Yes
Dynamic Destaging [23]	No	Yes
Pre-Partitioning (New)	No	Yes

Table 2.2: Overview of all six algorithms.

runs is larger than the number of available frames in memory, multiple levels of merging are needed (and new runs may be generated during the merging).

- Phase 2 (Group): Each output record of the last round of merging in Phase 1 (i.e., when the number of runs is less than or equal to the available frames in memory) will be aggregated on-the-fly, by keeping just one group as the current running group in memory and comparing the merge output record with the running group: if they have the same grouping key, they are aggregated; otherwise, the running group is flushed into the final output and replaced with the next merge output record. This continues until all records outputted from Phase 1 are processed.



(a) Sort in Phase 1 (b) Group in Phase 2

Figure 2.3: Memory structure in the Sort-based algorithm.

The algorithm uses only the available memory budget M, since (i) the in-place Quicksort algorithm [47] sorts $M-1$ frames with one frame as the output buffer, and (ii)

for merging, at most $M-1$ runs will be merged in a single merge round, and multiple-level merging will ensure this if the number of runs is larger than $M-1$. The group phase is pipelined with the last round of merging and it needs to maintain only one running group in memory (since the input records of this phase are provided in sorted order on the grouping key).

2.3.2 Hash-Sort Algorithm

The main disadvantage of the Sort-based algorithm is that it first scans and sorts the whole dataset. For a dataset that can be collapsed during aggregation, applying aggregation at an early stage would potentially save both I/O and CPU cost. The Hash-Sort algorithm that we developed for AsterixDB takes advantage of this observation by performing some aggregation before sorting. Figure 2.4 illustrates the workflow of this algorithm. Specifically, the Hash-Sort algorithm contains two phases, as described below:

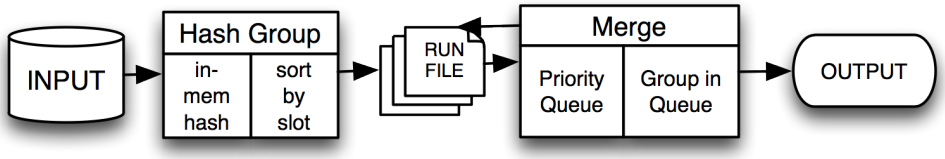
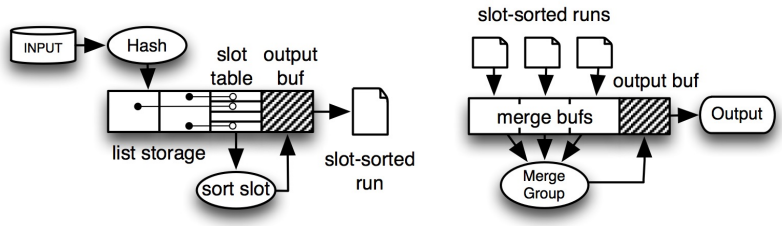


Figure 2.4: Hash-Sort Algorithm

- Phase 1 (Sorted Run Generation): An in-memory hash table is initialized using $M-1$ frames while the remaining frame is used as an output buffer. Input records are hashed into the hash table for aggregation. A new grouping key creates a new entry in the hash table, while a grouping key that finds a match is aggregated. When the hash table becomes full, the groups within each slot of the table are sorted (per slot) on the grouping key using in-place Quicksort, and the sorted slots are flushed into a run file in order of slot id (i.e., records in each run are

stored in (`slot-id`, `grouping key`) order). The hash table is then emptied for more insertions. This continues until all input records have been processed. If all groups manage to fit into the hash table, the table is then directly flushed to the final output (i.e., Phase 2 is not applicable).

- Phase 2 (Merge and Group): Each generated run is loaded using one frame as its loading buffer, and an in-memory loser-tree priority queue is built on the combination of (`slot-id`, `grouping key`) for merging and aggregation. The first group record popped is stored in main memory as the running group. If the next group popped has the same grouping key, it is aggregated. Otherwise, the running group is written to the output and is replaced by the new group (just popped). This process continues until all runs have been consumed. Similar to the Sort-based algorithm, at most $M-1$ runs can be merged in each round; if more runs exist, multiple-level merging is employed.



(a) Phase 1 (b) Phase 2

Figure 2.5: Memory structure in the Hash-Sort algorithm.

This algorithm also uses a bounded memory budget. Figure 2.5 shows the memory configuration in its two phases. In the first phase the in-memory hash table uses exactly $M-1$ frames of the memory, and the table is flushed and emptied when it is full. Sorting (although slot-based) and merging are similar to the Sort-based algorithm in terms of memory consumption.

2.3.3 Hybrid-Hash Variants

Hybrid-hash algorithms assume that the input data can eventually be partitioned so that one partition (the **resident partition**) can be completely aggregated in-memory, while each of the other partitions (**spilling partitions**) is flushed into a run and loaded back later for in-memory processing. I/O is thus saved by avoiding writing and re-reading the resident partition. Specifically, there are $(P + 1)$ partitions created, with the resident partition (typically partition 0) being aggregated in-memory using $M - P$ frames, and the other P partitions being spilled using P frames as their output buffers. The required number of spilling partitions P can be calculated for a given memory budget M assuming that (i) the full memory can contain an in-memory hash table for the resident partition plus one frame for each spilling partition, and (ii) the size of each spilling partition is bounded by the memory size (and can thus be processed in-memory in the next step). The following formula gives a formal description of this partition strategy:

$$\begin{aligned} M - P &= G * F - (M - 1) * P \\ \Rightarrow P &= \frac{G * F - M}{M - 2} \end{aligned} \tag{2.1}$$

where F is a fudge factor used to reflect the overhead from both the hash table and other structures (more about the fudge factor will be discussed in Section 2.5.7). This formula indicates that the total input data is processed as one resident partition (occupying $M - P$ frames for in-memory aggregation), and P spilled partitions (each will fit into memory using $M - 1$ frames). The above formula appeared in [49] for joins; we adapt it here for aggregation, so it uses the result set size G instead of the input size

R, because records from the same group will be aggregated and collapsed.

All hybrid-hash algorithms in this chapter process data recursively using two main phases as illustrated in Figure 2.6. In detail,

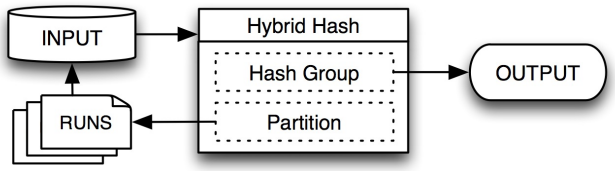


Figure 2.6: General Hybrid-hash algorithm structure.

- Phase 1 (Hash-Partition): If the input data is too large to be hybrid-hash processed in one pass ($G \geq M^2$), all memory is used to partition the input data into smaller partitions (**grace partitioning** in Grace Join [34]), and for each partition one run file is generated. Otherwise ($G < M^2$), partition 0 is immediately aggregated using an in-memory hash table. At the end of this phase, partition 0 will be either flushed into a run file (if its aggregation is not completed due to the incorrect estimation on partitioning) or directly flushed to the final output (otherwise).

- Phase 2 (Recursive Hybrid-Hash): Each run file generated above is recursively processed by applying the hash-partition algorithm of Phase 1. The algorithm terminates if there are no runs to be processed. To deal at runtime with grouping key value skew, if a single given run file's output is more than 80% of the input file that it was partitioned from, or the number of grace partitioning levels exceeds the number of the levels that would have been needed for the Sort-based algorithm, this particular run file will be processed next using the Hash-Sort algorithm (instead of recursive hybrid-hash) as a fallback to avoid deep recursion.

Clearly, hybrid-hash algorithms need G as an input parameter in order to man-

age memory space optimally. While the aim is to fully aggregate the resident partition in memory (when $G < M^2$), this is not guaranteed under strictly bounded memory by the existing hybrid-hash approaches we have seen, including the Original Hybrid-Hash [49], the Shared Hashing [50] and the Dynamic Destaging [22] algorithms. Thus in AsterixDB we propose a new approach using Pre-Partitioning that guarantees the completion of resident partition in memory. The details of these variants are described in the following subsections.

2.3.3.1 Original Hybrid-Hash

In this algorithm, adapted from [49], if an input record is hashed to partition 0, then it is inserted into the in-memory hash table for aggregation, otherwise it is moved to an output buffer for spilling. Figure 2.7 depicts the memory structure of this algorithm. Ideally partition 0 should be completely aggregated in-memory and directly flushed to the final output; however if the hash table becomes full, groups in the list storage area are simply flushed into a run (i.e., partition 0 also becomes a spilling partition).

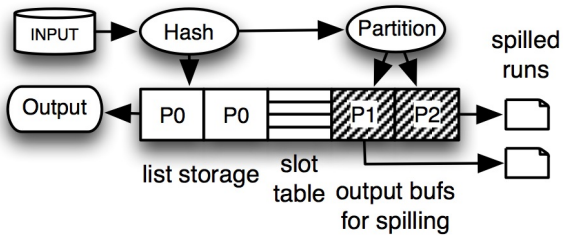


Figure 2.7: Memory Structure in the Original Hybrid-hash Algorithm.

Note that the proper choice of the number of spilled partitions P depends on the result size G which is unknown and can only be estimated. An incorrect estimation of G may result in partition 0 being too large to fit into memory and finally being spilled. While this may cause more I/O, the memory usage of this algorithm is still

tightly bounded, since at most M frames are used during the whole procedure.

2.3.3.2 Shared Hashing

The hybrid-hash algorithm proposed in [50] creates the same partitions as the Original Hybrid-Hash does, but the in-memory hash table is **shared** by all partitions. This sharing allows for aggregating data from both partition 0 **and** the other P partitions. Effectively, the Shared Hashing algorithm initially treats all partitions as 'resident' partitions. In order to use as much of memory for aggregation for all partitions, and also to reserve enough output buffers for spilling partitions, the list storage area of the hash table is divided into two parts: the **non-shared** part contains P frames for the P spilling partitions, while the remaining frames (**shared** part) are assigned to partition 0 but initially shared by all partitions. Using this layout, the P frames for spilling partitions can also be used for hashing and grouping before the memory is full, and then for spilling output buffers after that. Figure 2.8 illustrates the memory structure of this stage. `P1` and `P2` are the frames allocated to partition 1 and 2 respectively so they are not shared. Other frames (marked as `Px`) are assigned to partition 0, but also shared by partition 1 and 2 before any spilling. Spilling is triggered when a new group record arrives to one partition and there is no space available for more data from that partition (including the shared frames). The first two spillings are handled differently from future ones, as described below:

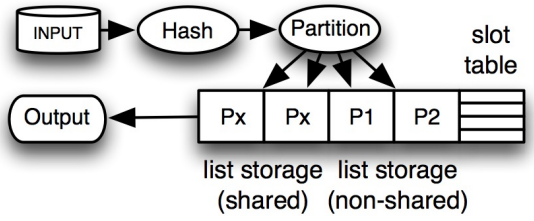


Figure 2.8: Memory structure before the first spilling of the Shared Hashing algorithm.

- First Spilling: When the first spilling is triggered (from any partition) by lack of additional space, all P spilling partitions are flushed. Each frame in the (soon-to-be) non-shared part is first flushed into a run for its corresponding partition using partition 0's output buffer. After flushing, the non-shared frames will become the output buffers for the P spilling partitions. Then the shared part is scanned. Group records from all spilling partitions are moved to the corresponding partition's output buffer for spilling, while groups of partition 0 are rehashed into a new list storage area built upon recycled frames (i.e., a frame in the shared part is recycled when all its records have been completely scanned and moved) and clustered together. Figure 2.9 depicts the memory structure when the scan is processed. After the first spilling the new list storage area belongs only to partition 0, and there is one output buffer for each spilling partition; the memory structure is now the same as the Original Hybrid-Hash showed in Figure 2.7.

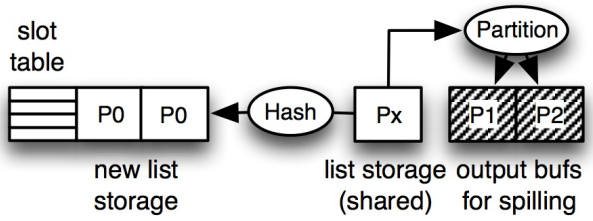


Figure 2.9: Memory structure during the first spilling of the Shared Hashing algorithm.

- Second Spilling: When the new list storage area has no more space for new group records from partition 0, partition 0 will be spilled. Its groups are flushed to a run file, and the frames they occupied are recycled. From now on, a single frame is reserved as the output buffer for partition 0 as well, and it is directly spilled like the other partitions.

The above algorithm uses bounded memory. Before the first spilling, the entire M-frame memory allocation is used as an in-memory hash table. When scanning the shared part in the first spilling, non-shared frames are reserved for the P spilling partitions, and frames recycled from the shared part are used for the new list storage area to cluster the partition 0 groups. After the second spilling the out-buffer frames for spilling partitions are obviously always memory-bounded.

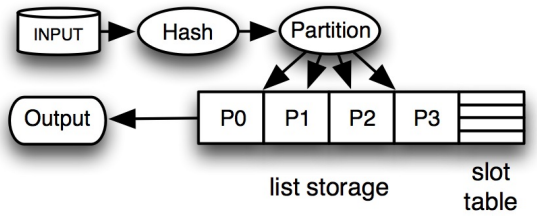
2.3.3.3 Dynamic Destaging

Unlike the previous two approaches, where the memory space for partition 0 is pre-defined (based on Formula 2.1), the Dynamic Destaging algorithm [22] dynamically allocates memory among all partitions, and spills the largest resident partition when the memory is full. After all records have been processed, partitions that remain in memory can be directly flushed to the final output (i.e., they are all resident partitions). This algorithm has two stages:

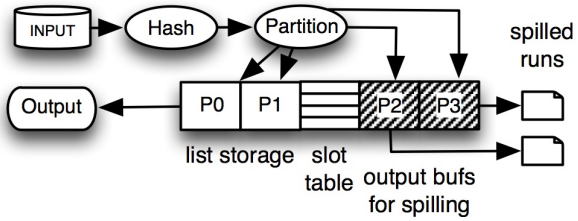
- Stage 1 (Initialization): An in-memory hash table is built so that one frame is reserved for the resident partition and each of the P spilled partitions in the list storage area, and the remaining frames are managed in a buffer pool. All partitions are initially considered to be resident partitions. Figure 2.10 (a) depicts the memory structure after this stage.

- Stage 2 (Hash-and-Partition): Each input record is hashed and aggregated into the frame of the corresponding partition. When a frame becomes full, a new frame is allocated from the pool for this partition to continue its aggregation. If no frame can be allocated, the largest (still) resident partition is spilled into a run file. Frames that this partition occupied are recycled, and a single frame

is now reserved as its output buffer. Additional records hashed to such a spilled partition will be directly copied to its output buffer for spilling (i.e., no aggregation happens for a spilled partition and no additional frames will be allocated for that partition in the future). Figure 2.10 (b) illustrates the memory structure after some partitions are spilled.



(a) Before spilling (all partitions are resident).



(b) After partition 2 and 3 are spilled.

Figure 2.10: Memory structure in the Dynamic Destaging algorithm.

Following [22], for computing the initial number of spilled partitions P, our implementation allocates between 50-80% of the available memory (i.e., 50% if the computed P value is less than 50% and 80% if the computed P is larger than 80%), in order to balance the size of the in-memory and spilling partitions (i.e., so the partition size is not too large or too small). Small runs created due to possible over-partitioning are merged and processed together in a single in-memory hash aggregation round, if the merged run can be fully processed in-memory (mentioned as **partition tuning** in [22]).

The Dynamic Destaging algorithm is memory bounded since memory is dynamically allocated among all partitions. When memory becomes full, a partition is spilled to recycle space. In the worst case, when all partitions are spilled, the available

memory can be dynamically allocated among all partitions and used simply as output buffers.

2.3.3.4 Pre-Partitioning

All of the approaches described so far assume that the hash function and the distribution of the hash values into partitions are properly chosen so that resident partition(s) can be completely aggregated in memory. Unfortunately, there can be no such guarantee, especially without precise knowledge about the input data. The naive approach of partitioning the hash value space based on Formula 2.1 will not work if the hash values used by the input data are not uniformly distributed in the hash value space. Moreover, these hybrid-hash aggregation algorithms are all derived from (and thus influenced by) hybrid-hash joins. One important property that distinguishes aggregation from join is that, in aggregation, the size of a group result is fixed and is not affected by duplicates. As a result, the memory requirement for a set of groups is fixed by the cardinality of the set (while the group size in a join could be arbitrarily large).

Based on these observations, we developed and implemented in AsterixDB the Pre-Partitioning algorithm. This algorithm divides the entire memory space similarly to the Original Hybrid-Hash, where $M - P$ frames are used for an in-memory hash table for partition 0. But, instead of assigning the groups of partition 0 based on hash partitioning, Pre-partitioning considers all groups that can be inserted into the in-memory hash table (before the table becomes full) as belonging to partition 0.

After the hash table is full, grouping keys that cannot be aggregated in the in-memory partition are spilled into the remaining P output frames. In order to decide whether a record should be spilled or aggregated, each input record needs a hash table

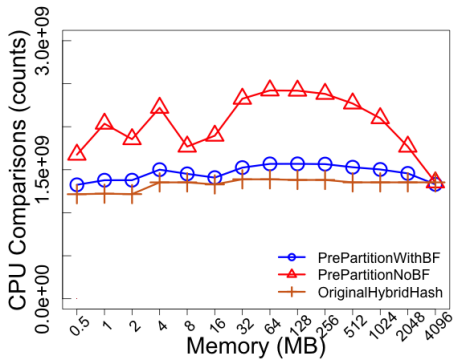
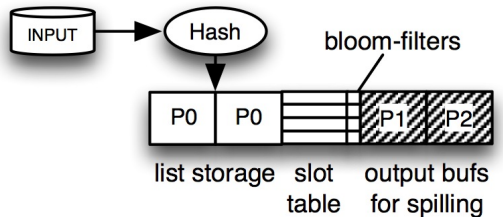


Figure 2.11: Comparisons of CPU cost among Pre-Partition with bloom filter, Pre-Partition without bloom filter, and the Original Hybrid-Hash.

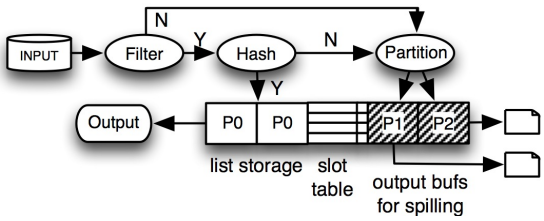
lookup to check whether it can be aggregated or not. This would cause a much higher hash lookup miss ratio compared with other hybrid-hash algorithms. To improve the efficiency of identifying the memory-resident vs. spilling groups, we add an extra byte as a mini bloom filter for each hash table slot. The bloom filter is updated when a new group is inserted into the slot (before the hash table becomes full). After the hash table is full, for each input record a lookup on the bloom filter is first performed, making a hash table lookup necessary only when the bloom filter lookup returns true. If the bloom-filter lookup returns false, it is safe to avoid looking into the hash table (since a bloom filter could only cause a false-positive error). For a properly sized hash table (i.e. where the number of slots is no less than the number of groups that can be contained in the table), the number of groups in each slot will be small (less than two on average), and a 1-byte bloom-filter per slot works well to reduce hash table lookups with a very low false-positive error rate. Figure 2.11 shows the CPU cost of aggregating 1 billion records with around 6 million unique groups using Pre-Partitioning with bloom filtering, Pre-Partitioning without bloom filtering, and the Original Hybrid-Hash algorithms. From the figure we can see that by applying the bloom filter, the CPU cost of the Pre-Partitioning algorithm is greatly reduced and becomes very close to the

cost of the Original Hybrid-Hash algorithm.

In order to reduce the overhead of maintaining the bloom filters, in our implementation no bloom filter lookup is performed before the hash table is full. This means that there is only the cost of updating the bloom filters when updating the hash table through a negligible bit-wise operation. This is because before the hash table is full, all records are inserted into the hash table anyway, and the benefit from bloom filters on reducing the hash misses is very limited (since a hash miss because of an empty slot can be easily detected without bloom filter lookup). Furthermore, if the dataset could be aggregated in memory based on the input parameters, no bloom filter will be needed, and the bloom filter overhead can be eliminated. Note that the output key cardinality (G in Formula 2.1) could be underestimated, and the bloom filters could be falsely disabled, causing more CPU cost on hash misses. Pre-Partitioning still outperforms other hybrid-hash algorithms in this case because other hybrid-hash algorithms have more extra I/O cost on spilling the in-memory partition. Section 2.5.6 shows our experiments in this scenario. Figure 2.12 shows the two stages of the Pre-Partitioning algorithm:



(a) Partition-0-build: before the hash table is full.



(b) Hash-And-Partition: after the hash table is full.

Figure 2.12: Memory Structure in the Dynamic Destaging Algorithm.

- Stage 1 (Partition-0-Build): Using Formula 2.1, P frames are reserved as the output buffers (to be used in Stage 2 for the spilling partitions). The remaining $M - P$ frames are used as an in-memory hash table storing groups of partition 0. Input records are inserted into this hash table for aggregation until the list storage area is full. If $P > 1$, a 1-byte bloom filter is used for each hash table slot, and all insertions to the hash table update the respective bloom filters. Figure 2.12 (a) shows the memory structure of this stage.

- Stage 2 (Hash-And-Partition): After the hash table is full, for each input record we check if that record has been seen before in partition 0 by first performing a bloom filter lookup; if the bloom filter lookup is positive, a hash table lookup follows, and it is aggregated if a match is found (no more memory is needed for this aggregation). Otherwise, this record is stored into one of the P output frames. When such a frame becomes full it is spilled. Figure 2.12 (b) illustrates this procedure. When all records have been processed, the groups aggregated in the in-memory hash table are directly flushed to the final output.

The Pre-Partitioning algorithm uses bounded memory since the in-memory hash table never expands beyond the $M - P$ pre-allocated frames. A benefit of this algorithm is that it allocates as many records to partition 0 as possible (until the in-memory hash table becomes full, at which time the pre-allocated $M - P$ frames are fully utilized) and this partition is guaranteed to be fully aggregated in-memory. Since the previous hybrid-hash variants cannot provide this guarantee, they may not fully utilize the pre-allocated memory for partition 0 (even if partition 0 could be finished in-memory).

We have also explored the idea of applying the bloom filter optimization to

Symbol	Description
b	Tuple size in bytes
o	Hash table space overhead factor (for its slot table and references of linked list)
p	Frame size in bytes
A	Collection of sorted run files generated
$\mathcal{D}(n, m)$	Dataset with n records and m unique keys
G	Output dataset size in frames
G_t	Number of tuples in output dataset
H	Number of slots in hash table
K	Hash table capacity in number of unique groups
M	Memory capacity in frames
R	Input dataset size in frames
R_t	Number of tuples in input dataset
R_H	Number of raw records inserted into a hash table before it becomes full

Table 2.3: Symbols Used in Models

other hash-based algorithms discussed in this chapter. However the overhead would be more significant than the benefit for the other algorithms. This is because a bloom filter is useful to avoid hash collisions (i.e. using a bloom-filter may avoid the hash lookup leading to a hash miss). However, with properly sized hash tables and assuming good hashing functions, most of the hash table insertions will not cause a hash collision, so the bloom filter does not help much reduce the collisions but introduces more memory overhead for the hash table.

2.4 Cost Models

We proceed by introducing applicable cost models for all six aggregation algorithms discussed in previous section. For simplicity we assume that the grouping keys are uniformly distributed over the input dataset. Moreover, for the hybrid-hash algorithm models it is assumed that the input parameters (size of input file, number of

unique keys etc.) are precise. The analysis focuses on the CPU comparison cost (for sorting and hashing) and the I/O cost (read and write I/Os). For simplicity, we omit the CPU and I/O costs for scanning the original input file and flushing the final result since they are the same for all algorithms (any may be pipelined). We also omit the pointer swapping cost in sorting and merging since it is bounded by the comparison cost (for a random dataset, the swap count is around 1/3 of the total comparisons [47]).

In our analysis, we use the following basic *component* models that are common for all algorithms, namely: the *input*, *sort*, *merge* and *hash* components. The details of these component models can be found in the Appendix. Table 2.3 lists the symbols used in the component and algorithmic models.

- **Input Component**: Let $\mathcal{D}(n, m)$ denote a dataset with total number of records n and containing m unique grouping keys. The model's input component computes the following two quantities:
  - $I_{key}(r, n, m)$, denotes the number of unique grouping keys contained in r records randomly drawn from $\mathcal{D}(n, m)$ without replacement (see Equation A.1).
  - $I_{raw}(k, n, m)$, denotes the number of random picks needed from $\mathcal{D}(n, m)$ in order to get k unique grouping keys (see Equation A.2).
- **Sort Component**: $C_{sort}(n, m)$ represents the number of CPU comparisons needed (using quicksort) in order to sort n records containing m unique keys (see Equation A.4).
- **Merge Component**: $C_{CPU.merge}(A, M)$ and $C_{IO.merge}(A, M)$ represent the CPU and I/O cost respectively, for merging a set

of files A using M memory frames (see Section A.3).

- **Hash Component**: Assume a hash table whose slot table has H slots and whose list storage area can store up to K unique keys (i.e. at most K unique groups can be maintained in the list storage area). The hash component computes the following quantities:

  - $H_{slot}(i, H, n, m)$ represents the number of occupied slots in a hash table with H slots after inserting i random records ($i \leq K$) taken from $\mathcal{D}(n, m)$ (see Equation A.6).

  - $C_{hash}(n, m, K, H)$ represents the total comparison cost until filling up the list storage area of a hash table with H slots and capacity K if the records are randomly picked from $\mathcal{D}(n, m)$ (see Equation A.9). Note that the hash table could become full before all records from $\mathcal{D}(n, m)$ are loaded.

  - $C_{hash}(n, m, K, H, u)$ is again the total comparison cost for filling up the list storage area as above, but assumes that $\mathcal{D}(n, m)$ has been partially aggregated, and the partially aggregated part (u unique records) are first inserted into the hash table before the random insertion.

2.4.1 Sort-based Algorithm Cost

The I/O cost for the Sort-based algorithm is solely due to external sorting since grouping requires just a single scan that is pipelined with merging. Let R denote the number of frames in the input dataset. The sort phase scans the whole dataset once using R write I/Os to produce A sorted runs (where $|A| = \frac{R}{M}$), each of size M, that are then merged. The total I/O cost is thus: $C_{IO} = R + C_{IO.merge}(A, M)$.

The CPU comparison cost C_{comp} consists of the sorting cost before flushing

the full memory into a run and the merging cost for merging all sorted runs. Hence:

$$C_{comp} = |A| * C_{sort}(R_{mem}, I_{key}(R_{mem}, R_t, G_t)) + C_{CPU.merge}(A, M) \quad (2.2)$$

where R_{mem} denotes the number of records that can fit in memory ($R_{mem} = \frac{Mp}{b}$, where p is the frame size and b is the input record size), R_t is the number of records in the input dataset, and G_t is the number of unique keys in the input dataset (which is the same as the number of tuples in the output dataset).

2.4.2 Hash-Sort Algorithm Cost

The Hash-Sort algorithm applies early aggregation using hashing and slot-based sorting. In the first phase (Sorted Run Generation), the I/O cost arises from flushing the unique keys in the hash table whenever it becomes full. Since the hash table uses the whole available memory M, its capacity is $K = \frac{Mp}{ob}$ (note that o is used to represent the memory overhead per record due to the hash table structure). The number of raw records inserted into the hash table until it becomes full is then: $|R_H| = I_{raw}(K, R_t, G_t)$. Once the hash table is full, all unique keys would be flushed after being sorted by (slot id, hash id). There are totally $\frac{R_t}{|R_H|}$ files generated, each file with size $\frac{Kb}{p}$; hence: $C_{IO.phase1} = \frac{R_t}{|R_H|} * \frac{Kb}{p}$.

The comparison cost for the first phase contains both hashing and slot-based sorting comparisons. The hashing comparison cost can be computed as

$$C_{comp.hash} = \frac{R_t}{|R_H|} * C_{hash}(|R_H|, G_t, K, H) \tag{2.3}$$

To estimate the sorting comparisons we note that when K unique keys have been inserted, the number of non-empty slots used is given by $H_u(|R_H|, H, R_t, G_t)$. Based on the uniform distribution assumption, the number of unique keys in each slot is: $L_{slot} = \frac{K}{H_u(|R_H|,H,R_t,G_t)}$. Since duplicates have been aggregated, the L_{slot} records to be sorted in each slot are all unique; hence the total number of comparisons due to sorting becomes:

$$C_{comp.sort} = H_u(|R_H|, H, R_t, G_t) * C_{sort}(L_{slot}, L_{slot}) \tag{2.4}$$

During the merging phase, the I/O cost includes the I/O for loading the sorted runs, and the I/O for flushing the merged file. The size of each sorted run generated by the sorting phase is the memory size M. The size of a merged file can be computed as the size (number) of unique keys contained in the sorted runs that are used to generate this merged file. The number of unique keys can be computed using the input component, given the number of raw records that are aggregated into the merged file. If A denotes the total sorted runs and A' denotes the files to be merged ($A' \subseteq A$ and $|A'| \leq M$), the number of raw records that will be aggregated into the merged file will be $\frac{R_t * |A'|}{|A|}$, so the number of unique keys in the merged file would be $I_{key}(\frac{R_t * |A'|}{|A|}, R_t, G_t)$. So the total I/O cost for merging the A sorted runs is

$$F(A') = |A'| * M + \frac{I_{key}(\frac{R_t * |A'|}{|A|}, R_t, G_t) * b}{p} \quad (2.5)$$

By applying $F(A')$ in $C_{IO.merge}(A, M)$, we can compute the total I/O cost for merging. The CPU comparison cost of merging the A run files $C_{comp.merge}(A, M)$ can be computed in a similar way using the merge component.

2.4.3 Hybrid-Hash Based Algorithm Costs

In this section we describe the cost model for the hash-partition phase (Phase 1) for each of the four hybrid-hash algorithms described in Section 2.3. In the recursive hybrid-hash phase (Phase 2), all algorithms recursively process the produced runs using their hash-partition algorithm, and their cost can be easily computed by simply applying the cost model from Phase 1 so we omit the details. When the key cardinality of the input dataset is too large for direct application of a hybrid-hash algorithm we need first to perform a simple partitioning until the produced partitions can be processed using hybrid hash. The cost of this partitioning is $2 * L * R$ for its I/O cost of loading and flushing, and $L * R_t$ for CPU cost of scanning, if L levels of partitioning are needed.

2.4.3.1 Original Hybrid-Hash

The Original Hybrid-Hash algorithm aggregates records from partition 0 only in its hash-partition phase while the other P partitions are directly spilled using P output buffers. Hence the available memory for the hash table is $(M - P)$ and the capacity of the hash table is $K = \frac{(M-P)p}{ob}$. Assuming that keys are uniformly distributed in the input dataset, partition 0 can be fully aggregated in the hash table. Since the

number of raw input records of partition 0 is $\frac{K}{G_t} * R_t$, the comparison cost for hashing is $C_{hash}(\frac{K}{G_t} * R_t, K, K, H)$ (since the $\frac{K}{G_t} * R_t$ records contain K unique keys). The I/O cost arises from loading the input records from the disk, and from spilling the raw records belonging to the P spilled partitions onto the disk; hence $C_{IO} = R + (R - \frac{K}{G_t} * R)$.

2.4.3.2 Shared Hashing

The uniform key distribution and precise input parameter assumptions made by our cost model eliminate the second spilling phase of the Shared Hashing algorithm; hence the following discussion concentrates on the first spilling phase. The Shared Hashing algorithm aggregates records from all partitions until the hash table is full. At this stage all memory except for one output buffer frame is used for the hash table, so the hash table capacity is $K = \frac{(M-1)p}{ob}$. The hash comparison cost is thus similar to the Hash-Sort algorithm, i.e., $C_{comp.before\_full} = C_{hash}(|R_H|, G_t, K, H)$.

During the first spilling, grouping keys of partition 0 that are already in the hash table are re-hashed in order to be clustered together in a continuous memory space. Remaining records of partition 0 are hashed and aggregated until the hash table becomes full again. The fraction of partition 0 (the resident partition) r_{res} and a spilled partition r_{spill} in the total input dataset can be computed based on Formula 2.1 as below:

$$r_{res} = \frac{M - P}{(M - P) + MP}, \quad r_{spill} = \frac{1 - r_{res}}{P}$$

The hash comparison cost after the first spilling (including re-hashing and inserting the remaining records from partition 0) can be computed by considering that the $r_{res} * K$ unique groups are inserted ahead:

$$C_{comp.after\_full} = C_{hash}(r_{res} * R_t - I_{raw}(r_{res} * K, R_t, G_t),$$
$$r_{res} * G_t, K, H, r_{res} * K) \quad (2.6)$$

where $I_{raw}(r_{res} * K, R_t, G_t)$ is the number of raw records inserted before the first spilling, while $r_{res} * K$ corresponds to the unique keys inserted before the first spilling that are then re-hashed during the first spilling. Here all partition 0 records are drawn from the $r_{res} * G$ unique keys assigned to partition 0.

After partition 0 is completely aggregated in memory and when the spilled runs are recursively processed, each run may already be partially aggregated, which corresponds to the 'mixed' input case. Hence the comparison cost for all resident partitions phase is computed as:

$$C_{comp.spill\_parts} = C_{hash}(r_{spill} * R_t -$$
$$I_{raw}(r_{spill} * K, r_{spill} * R_t, r_{spill} * G_t),$$
$$r_{spill} * G_t, K', H, r_{spill} * K) \quad (2.7)$$

This is very similar to the cost model showed in Equation 2.6, where the records inserted before the first spilling ($I_{raw}(r_{spill} * K, r_{spill} * R_t, r_{spill} * G_t)$) are collapsed into ($r_{spill} * K$) unique records and reloaded during the recursive hashing.

The I/O cost emanates from the spilling partitions only. Since part of each spilling partition has been aggregated before the table is full, the I/O cost contains the I/O both for spilling the partially aggregated partition, and for flushing the remaining raw records of that partition (computed by subtracting the aggregated raw records from

the total raw records of the spilling partition):

$$C_{IO.spill} = \frac{r_{spill} * K * b}{p} + r_{spill} * R - \frac{I_{raw}(r_{spill} * K, R_t, G_t) * b}{p} \tag{2.8}$$

where $\frac{r_{spill} * K * b}{p}$ is the I/O for spilling the partial aggregated results, and the remaining part is the I/O for spilling the raw records (where the records that are partially aggregated are excluded).

2.4.3.3 Dynamic Destaging

Until the hash table becomes full, the Dynamic Destaging algorithm behaves similarly to the Hash-Sort algorithm; hence the CPU comparison cost before any partition is spilled can be computed by $C_{hash}(|R_H|, G_t, K, H)$ (note that when this model is recursively applied to runs that have partially aggregated records, the 'mixed' input Equation A.13 should be used). When the hash table is full, the largest resident partition is spilled. The uniform assumption of the input dataset implies that at this time all partitions have the same number of grouping keys in memory; hence, any one partition can be randomly picked for spilling. If partition i is picked for the i-th spill, the total available memory for the hash table is $M - (i - 1)$ (where $i - 1$ frames are used as the output buffers for the spilled partitions). The number of in-memory aggregated groups of the i-th spilling partition can be computed using Formula 2.1 as:

$$K_i = \frac{K * (M - (i - 1))}{M(P + 1 - (i - 1))}$$

while the size of raw records hashed into the hash table for the i-th spilled partition is given by:

$$R_{H.i} = I_{raw}(K_i, \frac{R_t}{P+1}, \frac{G_t}{P+1})$$

Note here that for a specific partition i, the hash table capacity and the number of slots are the portion of the total K and H assigned to this partition. Then the CPU comparison cost for hashing this partition becomes:

$$C_{comp.i} = C_{hash}(R_{H.i}, K_i, \frac{K}{P+1}, \frac{H}{P+1}) \tag{2.9}$$

When spilling the i-th partition, since part of the partition has been hashed and collapsed before the partition is spilled, the total spilling I/O emanates from the raw records directly flushed ($\frac{R}{P+1} - R_{H.i}$), plus the partially aggregated unique keys ($\frac{K_i * b}{p}$); hence:

$$C_{IO.i} = \frac{K_i * b}{p} + \frac{R}{P+1} - R_{H.i} \tag{2.10}$$

This cost is summed for all spilled partitions. The number of spilled partitions, P_s, can be estimated by the following inequality (inspired by Formula 2.1), where the remaining $P + 1 - P_s$ partitions have enough memory to be completely aggregated in memory:

$$\frac{G}{P+1} \leq \frac{K * (M - P_s)}{M(P + 1 - P_s)} \tag{2.11}$$

2.4.3.4 Pre-Partitioning

The Pre-Partitioning algorithm aggregates records from partition 0 only in its hash-partition phase, while the other P partitions are directly spilled using P output buffers. When bloom filters are used with the hash table slot headers, there is an overhead of one byte per slot, or formally $o' = o + \frac{1}{b}$. The capacity of the list storage area is thus $K = \frac{(M-P)p}{ob+1}$. Since the algorithm guarantees that partition 0 can be fully aggregated in the hash table, the number of raw input records of partition 0 is $\frac{K}{G_t} * R_t$. The I/O cost consists of loading the records to be processed and spilling the raw records in the P spilled partitions, i.e.:

$$C_{IO} = R + (R - \frac{K}{G_t} * R) \tag{2.12}$$

The CPU comparison cost includes the cost of hashing the records of partition 0 into the hash table, plus the cost for checking whether a record should be spilled (for records from the P spilling partitions). Assume that the per-slot bloom-filter has a false positive ratio α. Then for each of the $(R_t * (1 - \frac{K}{G_t})$ spilled records, if the bloom filter can detect that the record is not in the hash table, the record is directly flushed (we omit the bloom filter lookup cost since it is negligible compared with the hash comparison cost). If the bloom filter fails to detect that the record is not in the hash table (false positive error with probability α), a hash table lookup for the record will cause a hash miss with cost of $\frac{K}{H}$. Therefore the CPU cost is given by:

$$\begin{aligned} C_{comp} =& C_{hash}(\frac{K}{G_t} * R_t, K, K, H) \\ &+ \alpha(R_t * (1 - \frac{K}{G_t}) * \frac{K}{H}) \end{aligned} \tag{2.13}$$

2.5 Experimental Evaluation

We have implemented all algorithms as operators in the Hyracks platform [8] and performed extensive experimentation. The machine hosting Hyracks is an Intel Xeon E5520 CPU with 16GB main memory and four 10000 rpm SATA disks. We used the Java 6 software environment on 64-bit Linux with kernel version 2.6.18-194.el5. We ran the example query of Section 2.2.1 on a synthetic UserVisits dataset (table) that has two fields: a string `ip` field as the grouping key, containing an abbreviated IPv6 address (from 0000:0001::2001 to 3b9a:ca00::2001 for 1 billion records), and a double `adRevenue` field (randomly generated in $[1, 1000]$). To fully study the algorithm performance and validate the cost models, we consider the variables listed below. The values that we used for these variables in our experiments (organized by subsection) appear in Table 2.4.3.3.

- *Cardinality ratio:* the ratio between the number of raw input records (input size R) and the number of unique groups (output size G).
- *Memory:* the size of the memory assigned for the aggregation.
- *Data distribution:* the distribution of the groups (keys) in the dataset.
- *Hash table slots:* the number of slots in the hash table, measured by the ratio between the number of slots and the number of unique keys that can fit in the list storage area.
- *Fudge factor:* the hybrid-hash fudge factor.
- *Unique Group Estimation Error:* (applies only to hybrid-hash algorithms) the ratio between the user (query compiler) specified and the actual number of unique groups.

2.5.1 Cost Model Validation

To validate the accuracy of our models, we depict the I/O and CPU (as predicted by the models and measured by the experiments) of the six algorithms in different memory configurations for two datasets with cardinality ratios 100% and 0.02% in Figures 13 and 14 respectively; we also experimented with cardinalities 44.1% and 6.25% which showed similar behavior (not shown due to the space limitation). As we can see, our models can predict both the I/O and the CPU cost with high precision. In particular, the I/O cost estimation is consistently very close to the actual I/O. For most cases, the cost for the (hash) CPU comparisons is slightly underestimated by our models because they assume no skew; however, in reality even slightly skewed data will result in higher hash collisions. This explains the slightly lower model prediction for the CPU cost of the hash-based algorithms in Figure 2.14.

There are cases where our models overestimate the CPU cost, as when processing the "all unique" dataset (Figure 2.13, with cardinality ratio 100%) for the Dynamic Destaging and Shared Hashing algorithms. This is because with actual data, the hash table spilling could be triggered earlier than the model prediction since the key distribution is not perfectly uniform; as a result, less groups from spilling partitions are hashed into the dynamic/shared hash scheme, leading to less actual CPU cost.

Among all algorithms, the CPU model for Dynamic Destaging showed the largest overestimation compared to the real experiments for some configurations. The reason is that in these cases, our cost model assumes that the resident partition can be completely aggregated in-memory, however in reality our experiments show that in these configurations, the resident partition has also been spilled due to the imperfect hash partitioning and dynamic destaging technique (i.e., evicting the right partitions for

spilling) in reality. When reloading the spilled resident partition, the number of hash table collisions is less since the records are hashed to the whole hash table space instead of just a potion of it, so the actual CPU comparison cost is less than predicted.

2.5.2 Effect of Memory Size

To study the effect of memory size on the aggregation algorithms we measured their running time using the four uniform data sets (with cardinality ratio 100%, 44.1%, 6.25% and 0.02%) in different memory configurations (0.5M to 4G). (The effects of skewed data are examined later). In Figure 2.15, we show the running time, CPU comparison cost and I/O cost for all these experiments. When considering the CPU cost, the algorithms that use sorting require more CPU than the pure hashing algorithms. The I/O cost for all algorithms decreases when memory increases (since more records can be aggregated in memory).

We first observe that for larger memories (memory larger than 64M) the running time of the Sort-based algorithm increases. This is because larger memory settings cause higher cache misses for the comparing and swapping in the sorting procedure. Furthermore, when the cardinality ratio is high (100%, 44.1%, and 6.25%), the total CPU cost for sorting is increasing according to Formula A.4 of the sort component in Appendix A.2(the records to be sorted in each full memory chunk m is larger). This can also be observed through the similar rising of the CPU cost for memories larger than 64M (Figure 2.15 (e-h)). Thus it is not always the case that larger memory leads to better performance in the Sort-based algorithm. Different from the Sort-based algorithm, the Hash-Sort algorithm has better performance when the memory is larger because it utilizes collapsing, and most of the time it is faster than the Sort-based algorithm (except for the case with small memory, where the collapsing cannot be fully exploited).

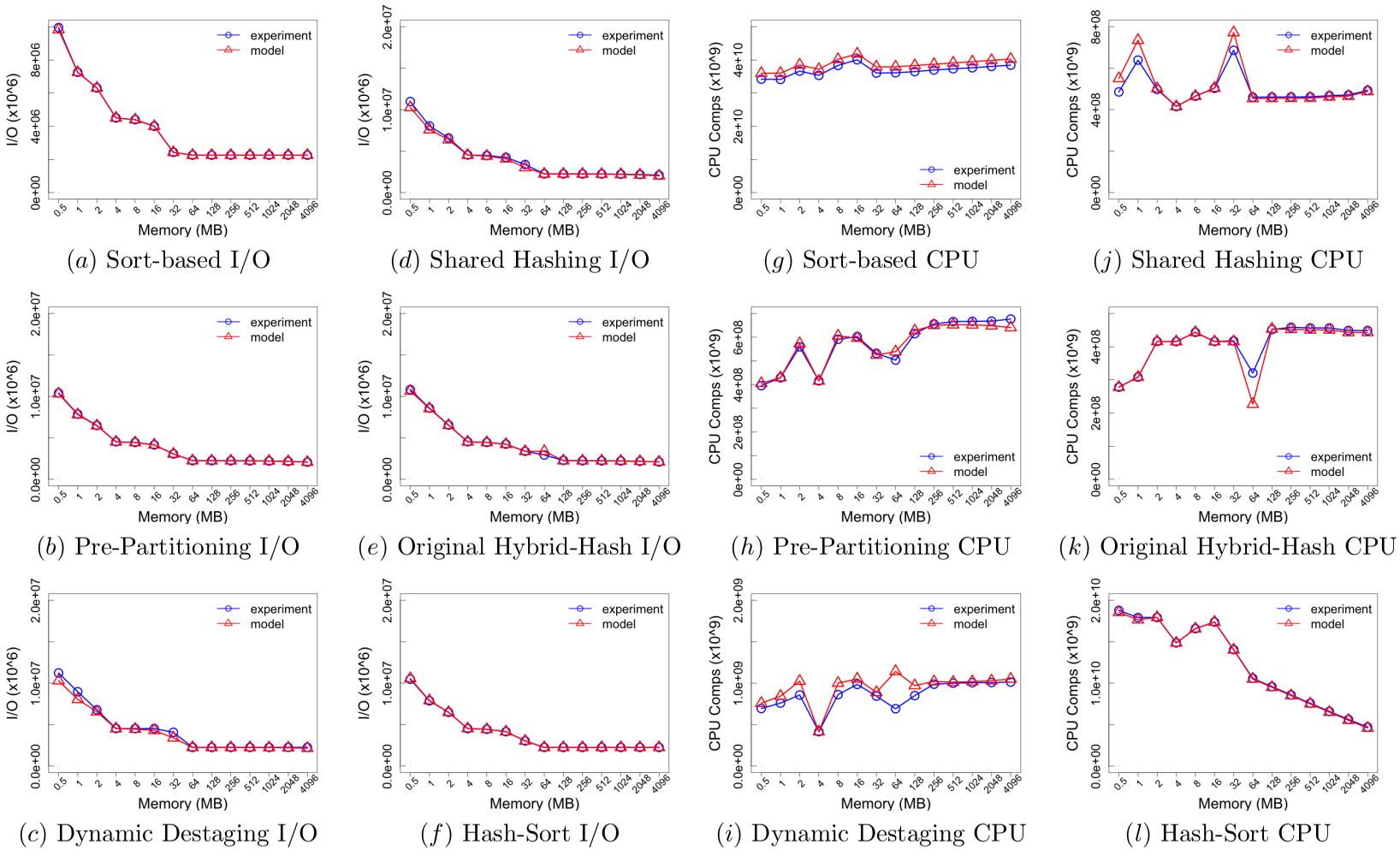
The four hybrid-hash algorithms have the best performance since they avoid sorting and merging. Among the hybrid-hash algorithms, the Pre-Partitioning algorithm has the most robust performance along all memory and key cardinality configurations. This is because Pre-Partitioning always creates the resident partition to fill up the in-memory hash table. This will reduce both the I/O (since more groups are aggregated within the resident partition) and the CPU comparison cost (since less spilled records need to be processed recursively). Furthermore, by using bloom filters within the hash table, the extra cost for hash misses is reduced so its CPU cost is just slightly higher than the Original Hybrid-Hash algorithm (as showed in Figure 2.11).

Also note that according to Formula 2.1, the memory space reserved for the resident partition $(M - P) = \frac{M^2-3M-G*F}{M-2}$ is not linearly associated with the memory size. This means that when the memory increases, although the number of hash table slots increases correspondingly, the size of the resident partition does not increase linearly. So the hash collision could vary based on the ratio between the unique records in the resident partition and the hash table slots. In the case that this ratio is higher due to a larger increase of the unique records in the resident partition than the increase of the hash table slots, there will be more hash comparison cost for aggregating the resident partition. This explains the spikes of the CPU cost for all hybrid-hash algorithm along different memory configurations.

We further notice that the running time for Dynamic Destaging is increasing (it becomes larger than the other hybrid-hash algorithms) for memories between 16M and 2048M. In these memory configurations only one round of hybrid-hash is needed (i.e., no grace partition is used). However the partition tuning optimization [22] increases the number of partitions as the memory increases, which causes more cost overhead for maintaining the spilling files. Furthermore, as the memory increases, the number

of records from spilling partitions that have been partially aggregated and flushed will be larger (recall that in Dynamic Destaging, spilling partitions are dynamically spilled in order to maximize the in-memory aggregation); this could potentially increase the hashing cost because all partial results must be reloaded and hashed again.

Finally when the memory size is relatively very large (4G), all hybrid-hash algorithms have the same running time, as no spilling happens (so all can do in-memory aggregation).



(a) Sort-based I/O (d) Shared Hashing I/O (g) Sort-based CPU (j) Shared Hashing CPU

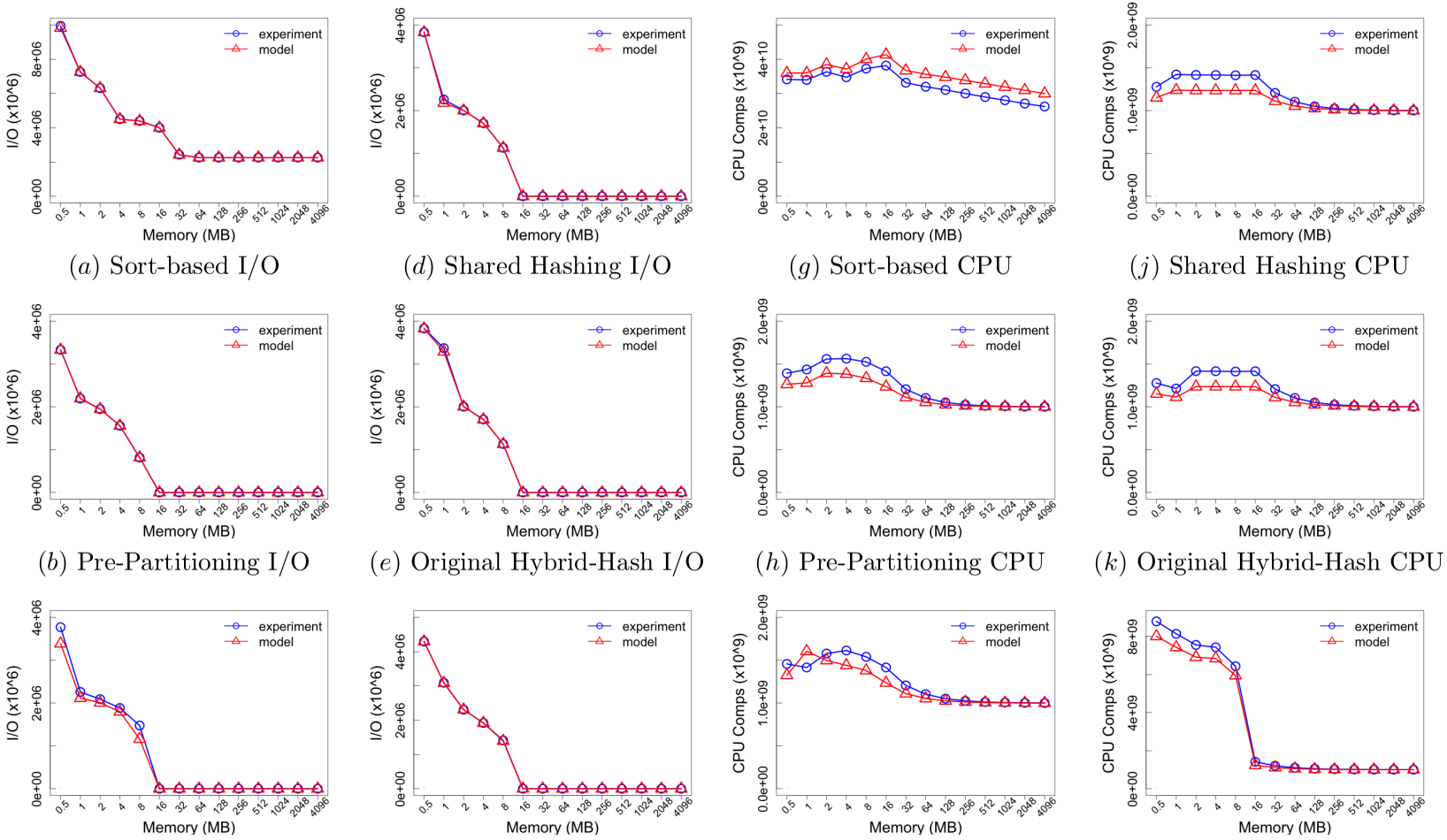
(b) Pre-Partitioning I/O (e) Original Hybrid-Hash I/O (h) Pre-Partitioning CPU (k) Original Hybrid-Hash CPU

(c) Dynamic Destaging I/O (f) Hash-Sort I/O (i) Dynamic Destaging CPU (l) Hash-Sort CPU

Figure 2.13: Model validation (100% cardinality ratio).

Subsection	Cardinality	Memory	Distribution	HT Slots	Fudge	HH Error
2.5.1, 2.5.2, 2.5.3	100%, 44.1%, 6.25%, 0.02%	0.5M $\sim$ 4G	Uniform	1	1.2	1
2.5.5	6.25%	1M, 64M, 4G	Uniform	1	1.2	1
2.5.6	0.02%	4M, 16M	Uniform	1	1.2	4096 $\sim$ 1/4096
2.5.4	1%	2M $\sim$ 128M	Uniform, Zipfian, Self-Similar, Heavy-Hitter, Sorted	1	1.2	1
2.5.7 (hash table slot)	6.25%	2M, 4G	Uniform	1, 2, 3	1.2	1
2.5.7 (fudge factor)	6.25%	2M, 4G	Uniform	1	1.0 $\sim$ 1.6	1

Table 2.4: Performance related factors used in the experimental evaluation.

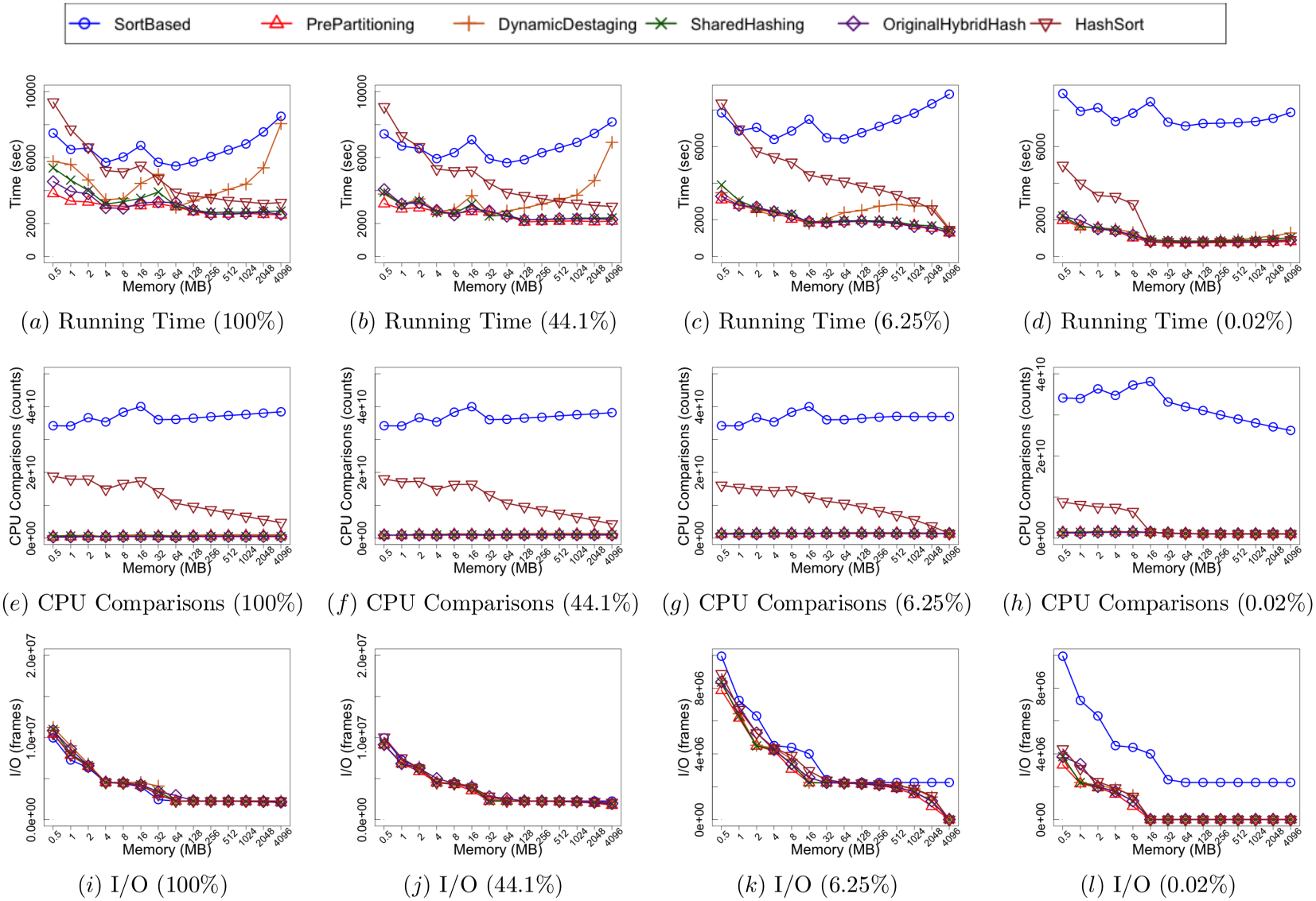


(a) Sort-based I/O (d) Shared Hashing I/O (g) Sort-based CPU (j) Shared Hashing CPU

(b) Pre-Partitioning I/O (e) Original Hybrid-Hash I/O (h) Pre-Partitioning CPU (k) Original Hybrid-Hash CPU

(c) Dynamic Destaging I/O (f) Hash-Sort I/O (i) Dynamic Destaging CPU (l) Hash-Sort CPU

Figure 2.14: Model validation (0.02% cardinality ratio).

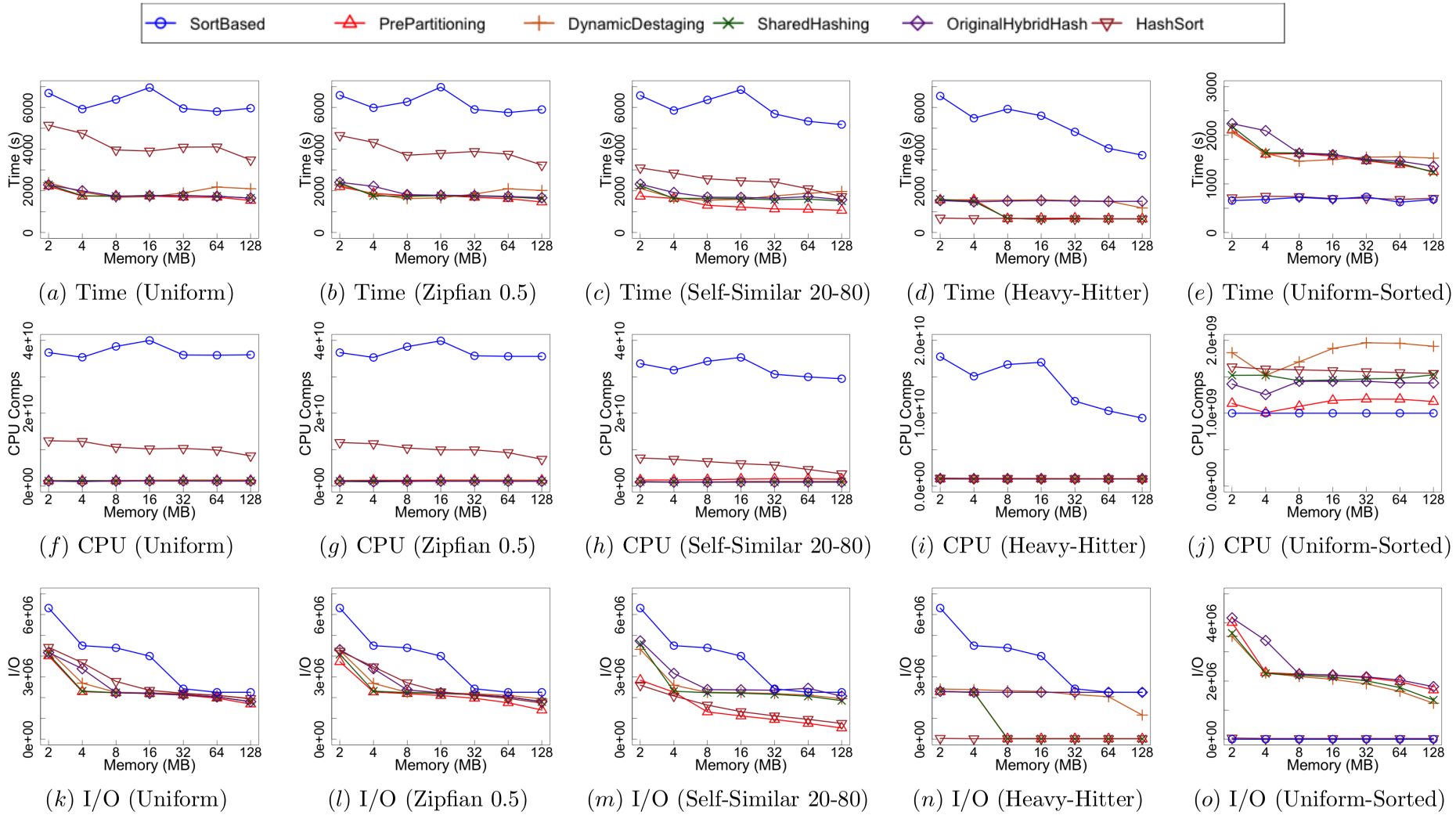


(*a*) Running Time (100%) (*b*) Running Time (44.1%) (*c*) Running Time (6.25%) (*d*) Running Time (0.02%)

(*e*) CPU Comparisons (100%) (*f*) CPU Comparisons (44.1%) (*g*) CPU Comparisons (6.25%) (*h*) CPU Comparisons (0.02%)

(*i*) I/O (100%) (*j*) I/O (44.1%) (*k*) I/O (6.25%) (*l*) I/O (0.02%)

Figure 2.15: Experiments with different cardinality ratios and memory sizes.



(a) Time (Uniform) (b) Time (Zipfian 0.5) (c) Time (Self-Similar 20-80) (d) Time (Heavy-Hitter) (e) Time (Uniform-Sorted)

(f) CPU (Uniform) (g) CPU (Zipfian 0.5) (h) CPU (Self-Similar 20-80) (i) CPU (Heavy-Hitter) (j) CPU (Uniform-Sorted)

(k) I/O (Uniform) (l) I/O (Zipfian 0.5) (m) I/O (Self-Similar 20-80) (n) I/O (Heavy-Hitter) (o) I/O (Uniform-Sorted)

Figure 2.16: Experiments on skew datasets.

2.5.3 Effect of Cardinality Ratio

Figure 2.15 also compares the algorithms for different cardinality ratios. Note that full in-memory aggregation happens for the 0.02% dataset when memory is larger than 16M, while it occurs for the 6.25% dataset only for the 4G memory. In the higher cardinality ratio datasets (100% and 44.1%) there is no in-memory aggregation (since the number of grouping keys is so large that all algorithms need to spill).

The Sort-based algorithm is typically slower than the rest, with the hybrid-hash algorithms being the fastest and the Hash-Sort falling in between (except for the case of very small memories and high cardinalities to be discussed below). As the cardinality ratio increases, the gap in performance between the Sort-based and the hybrid-hash algorithms is reduced. This is because a higher cardinality means more unique keys, and thus less collapsing, which reduces the advantage of hashing. Note that when the memory is small and the cardinality is large, the Hash-Sort algorithm performs even worse than the Sort-based algorithm because there is very limited benefit from early aggregation and the hash cost is almost wasted.

The hybrid-hash-based algorithms are greatly affected by the higher cardinality ratio, as fewer records can be collapsed through aggregation and the performance mainly depends on the effectiveness of partitioning. The spikes in the CPU cost (caused by the non-linear correlation between the resident partition size and the hash table size; see the discussion in the previous subsection) are more clear for data sets with higher cardinality ratios since the hash miss cost is more significant.

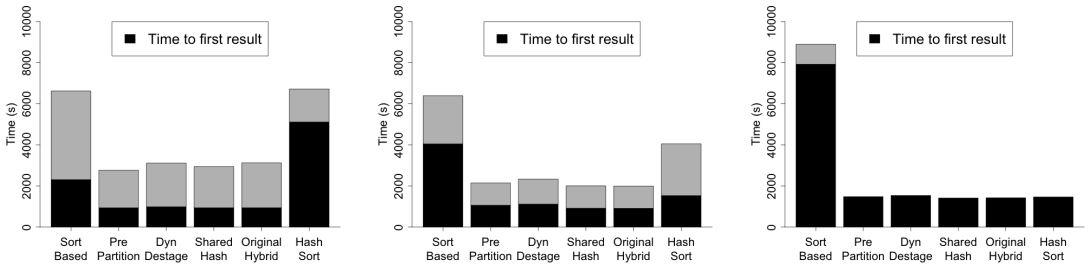
2.5.4 Aggregating Skewed Data

To examine the performance of the algorithms when aggregating skewed data, we considered the following skewed datasets, each with 1 billion records and 10 million unique keys, generated using the algorithms described in [25]:

- *Uniform*: all unique keys are uniformly distributed among the input records;
- *Zipfian*: we use skew parameter 0.5;
- *Self-similar*: we use the 80-20 proportion;
- *Heavy-hitter*: we choose one key to have $10^9 - (10^7 - 1)$ records, while all other keys have only one record each;
- *Sorted uniform*: we use a uniform data set with records sorted on the grouping key.

Figure 2.16 shows the running time, CPU cost and I/O cost of all algorithms for different skew distributions. Overall we observe that if the skew distribution is similar to the uniform distribution (the Zipf and the Self-Similar data sets), the behaviors of the algorithms are similar to the uniform case. A common characteristic of the two less-skewed datasets (Zipf and Self-Similar) is that the duplicates are distributed in a "long-tail" pattern. There are a few keys with very many duplicates (the peak of the distribution) and many keys with very few duplicates (the tail part). Nevertheless, statistically the peak in the Zipf dataset is lower than the peak in the Self-Similar dataset and its long-tail part is higher than the long tail of the Self-Similar dataset. Since there are more duplicates per key in the Zipf dataset, more hash comparisons are needed.

For the Zipf and Self-Similar datasets, the Hash-Sort algorithm is overall slower than the hybrid-hash based algorithms because (1) these datasets are not sorted, so the



(*a*) 1M memory (*b*) 64M memory (*c*) 4G memory

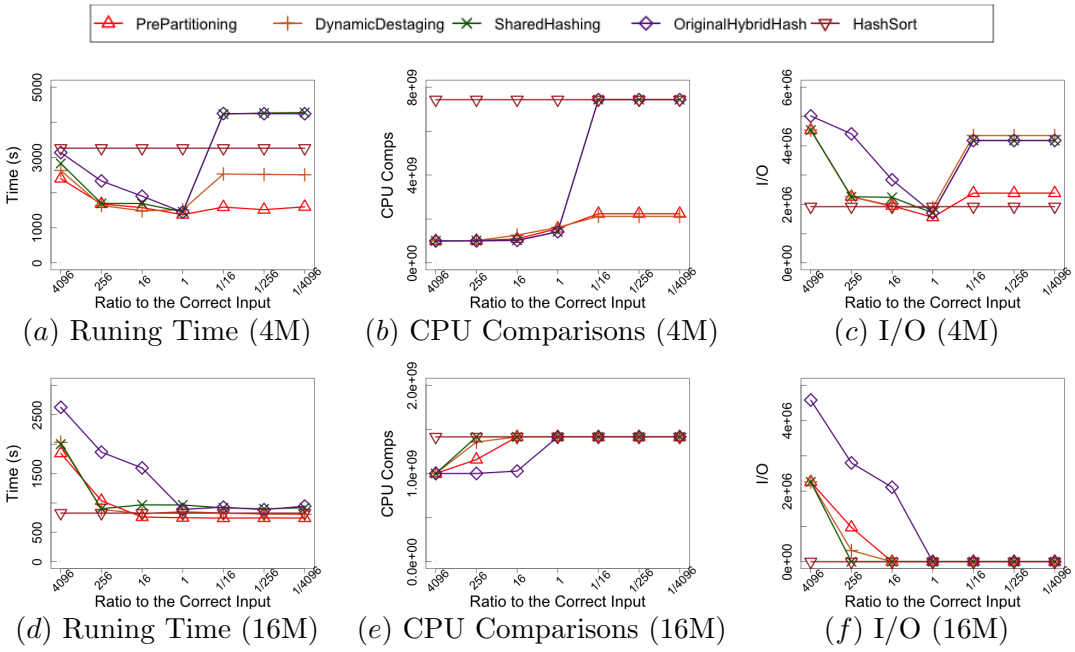
Figure 2.17: Time to the first result as part of the total running time.

Hash-Sort algorithm needs to sort and merge the intermediate results, and (2) since more grouping keys have duplicates, the same grouping key could be in multiple run files, which further increases the run file size and the cost for merging. For these datasets, the Pre-Partitioning algorithm has the best running time since it always fills up the memory space reserved for the resident partition, so more groups can be collapsed into the resident partition. This greatly reduces the total I/O cost for the Pre-Partitioning algorithm compared with other hybrid-hash algorithms, leading to a lower running time. It is interesting to note that this behavior is more apparent in the Self-Similar than the Zipf dataset. This is because the tail part in the Self-Similar dataset is smaller, so the size of the spilling partitions would be smaller when compared with the Zipf dataset. This will reduce both the hash miss cost for checking the spilling records and the I/O cost for spilling partitions.

For the Heavy-Hitter and the Uniform-Sorted datasets, the nature of their skew is more significant compared with the uniform case, so their behaviors are quite different than the uniform case. In particular, for the Heavy-Hitter data set, the Hash-Sort algorithm has the best overall performance. The algorithm collapses many duplicates in this data set in its early aggregation; moreover, its slot-based sorting strategy can minimize the sorting cost for merging. The Original Hybrid-Hash algorithm performs the worst in this case because the partition containing the heavy hitter key contains

99% of the total records; this causes the algorithm to fallback to the Hash-Sort (because it has more than 80% of the original input content as mentioned in Section 2.3.3). The Dynamic Destaging algorithm also performs badly due to the fallback, but the fallback is triggered by partition tuning. This is because partitions that do not contain the heavy hitter key are underestimated on their grouping key cardinality, and partition tuning merges them based on the underestimated cardinality. After merging is done, the key cardinality is greater than the memory capacity so these partitions are spilled again. Finally all spilled partitions are processed through the fallback algorithm (the hybrid-hash level is deeper than a Sort-based algorithm), resulting in longer running time. The Shared Hashing algorithm performs better when grace partitioning is not needed because it collapses the partition containing the heavy hitter by maximizing the in-memory aggregation through the shared hash table. A similar effect happens for the Pre-Partitioning algorithm, but it performs better since it always guarantees that the resident partition can be completely aggregated in memory.

For the uniform-sorted dataset, the Sort-based algorithm performs the best since it only needs a single scan over the sorted data to finish the aggregation. The Hash-Sort algorithm still shows good running times because it can aggregate each group completely in the sorted run generation phase, utilizing the sort order. However it is slightly slower than the Sort-based algorithm due to its higher I/O cost (because of the overhead of the hash table) and the CPU cost (since hashing is more expensive than the sequential match-and-scan procedure). The four hybrid-hash algorithms perform worse because all partitions produced by grace partitioning (for the 2M and 4M memory) or by the hybrid-hash algorithms (for 8M or larger memory) have to be processed by a recursive hybrid-hash procedure. With 4M memory, the Original Hybrid-Hash performs worse than the other hybrid-hash algorithms because they have better hash collapsing



(a) Runing Time (4M) (b) CPU Comparisons (4M) (c) I/O (4M)

(d) Runing Time (16M) (e) CPU Comparisons (16M) (f) I/O (16M)

Figure 2.18: Sensitivity on input error for Hybrid-Hash algorithms

effect; as a result, they can finish the hybrid hash aggregation one level earlier than the Original Hybrid-Hash algorithm using less I/O.

2.5.5 Time to First Result (Pipelining)

To check whether these algorithms can be pipelined effectively, we measure the time needed to produce the first aggregation result as another aspect of their performance. Figure 2.17 depicts the results using the 6.25% dataset in three different memory configurations. The full bar height corresponds to the total running time (full aggregation), while the bottom solid part corresponds to the time until the first aggregation result is produced. The earlier the aggregation result is produced, the better the algorithm can fit into a pipelined query stream.

For the hybrid-hash algorithms, the solid part in Figure 2.17 includes the time for grace partition, and the time for processing the resident partition in memory, while the gray part represents the time for recursive aggregation of the spilled partitions.

Blocking in the hybrid-hash algorithms occurs mainly due to the aggregation of the resident partition. For larger memory sizes, the resident partition is larger, so it takes more time to aggregate all records of the resident partition, resulting in slightly longer times to first result for the hybrid-hash algorithms. For very large memory (4G) there is no grace partitioning, and since all records are in memory, they need to be fully aggregated before the first result is produced; thus the time to first result is also the time when the full aggregation is completed.

For both the Sort-based and the Hash-Sort algorithms, the solid part includes the time for generating sorted runs plus the time for merging sorted runs until the final merging round. The gray part indicates the time for the last merging phase, where the aggregation results are produced progressively during merging. As the memory size increases, the time to first result for the Sort-based algorithm increases because the time for merging is longer. For very small memory (1M) the Hash-Sort algorithm experiences a longer blocking time because it uses both hashing and sorting, while the hashing does not collapse many records. As memory increases, the hashing becomes more effective in collapsing which reduces both the sorting and merging time. For very large memory (4G), the Hash-Sort aggregates all records in memory and thus the time to first result is also the time to full aggregation (similarly to the hybrid-hash algorithms).

2.5.6 Input Error Sensitivity of Hybrid Hash

The performance of all hybrid-hash algorithms is closely related to the input key cardinality G. Note that G serves as an exploit input of the hybrid-hash algorithm, as it is used to compute the number of partitions P. In practice the input set is not known in advance, so we estimate G. Since such estimation may not be accurate, we also tested the performance of the hybrid-hash algorithms assuming that G is over/under-

estimated. Using the dataset with cardinality ratio 0.02%, we ran experiments where P was computed assuming various (incorrect) values for G. In particular, we varied G from a far over-estimated ratio (4096 times the actual cardinality) to a quite underestimated ratio (1/4096 of the actual cardinality). Figure 2.18 shows the experimental results for two different memory budgets (4M and 16M). When the input parameter is correct (i.e., the ratio is 1), the first memory configuration causes spills whereas the second memory configuration can be processed purely in memory. We also depict the running time of the Hash-Sort algorithm for comparison (since Hash-Sort does not depend on the parameter G).

Our experiments show that both overestimation and underestimation can affect the performance of the hybrid-hash algorithms. Specifically, an overestimation will cause unnecessary grace partitioning, and will thus increase the total I/O cost. In the worst case all hybrid-hash algorithms do grace partitioning, causing slower running times than the Hash-Sort algorithm. An underestimation will falsely process the aggregation earlier, resulting in less collapsing in the hybrid-hash and further grace partitioning.

More specifically, the results in Figure 2.18 show that the Shared Hashing algorithm and the Original Hybrid-Hash may fallback to the Hash-Sort algorithm if the partition size is underestimated and turns out to be too large. The Dynamic Destaging algorithm works well in the underestimation case, as it always uses at least 50% of the available memory for partitioning. Among all hybrid-hash algorithms, Pre-Partitioning achieves better tolerance to the error in the grouping key cardinality G; this is due to its guarantee that the in-memory partition will be completely aggregated. Pre-Partitioning has more robust performance for underestimated cases since it can still guarantee the complete aggregation of the resident partition, and it can also gather some statistics while aggregating the resident partition. It can then use the obtained statistics to guide

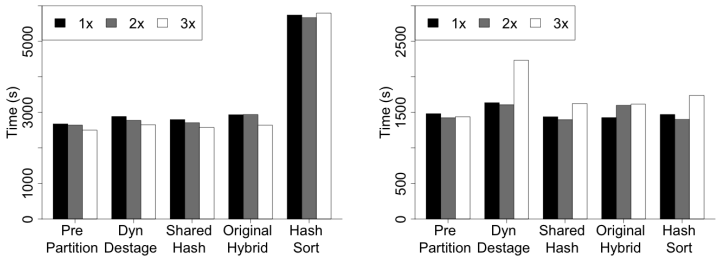
the recursive processing of the spilled partitions.

2.5.7 Hash Implementation Issues

During the implementation of the hash-based algorithms (all four hybrid-hash algorithms, and also the Hash-Sort algorithm) we faced several issues related to the proper usage of hashing. Considering the quality of the hash function, we used Murmur hashing [2]. We tried the multiplication method [35] (the default hashing strategy in Java) in our experiments, but we found that its hash collision behavior deteriorated greatly for the larger grouping key cardinalities in our test datasets. Another issue related to the usage of the notion of hash function family for the hybrid-hash algorithms. It is important to have non-correlated hash functions for the two adjacent hybrid-hash levels. In our experiments we used Murmur hashing with different seeds for the different hybrid-hash levels.

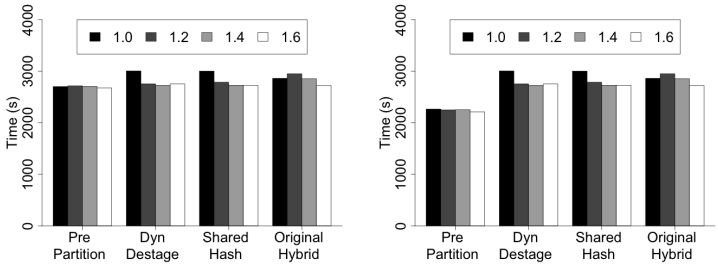
We also examined how the hash table size (slot table size, or the number of slots in the slot table) affects performance. Given a fixed memory space, an in-memory hash table with a larger number of slots (which could potentially reduce hash collisions) in its slot table would have a smaller list storage area (so a smaller hash table capacity). Thus, the number of slots should be properly picked to trade-off between the number of hash collisions and the hash table capacity. In literature, it is often suggested to use a slot table size that is around twice the number of unique groups that can be maintained in the list storage area. Figure 2.19 depicts the running times of the hash-based algorithms with varying slot table sizes (set to be 1x, 2x and 3x the number of unique groups maintained). In the small memory case, different slot table sizes do not affect the total running time significantly. In the larger memory case, all hash-based algorithms can aggregate the data in-memory when the slot table size is 1x (equal to

the number of unique keys). Most algorithms do in-memory aggregation except for the Original Hybrid-Hash, which spills due to the larger slot table overhead. When the slot table size is 3x, only the Pre-Partitioning algorithm can complete the aggregation in-memory, because it always fills up the memory for resident partition before trying to spill. (In all other experiments we picked 1x so that all hash-based algorithms can finish in-memory for 4G memory).



(*a*) Hash table size (2M) (*b*) Hash table size (4G)

Figure 2.19: Running time with different hash table sizes (as the ratios of number of slots over the hash table capacity).



(*a*) Fudge factor (2M) (*b*) Fudge factor (4G)

Figure 2.20: Running time with different fudge factors.

Finally, we also explored the importance of the fudge factor F in the hybrid-hash algorithms. This factor accounts for the extra memory overhead including both **hash table overhead** (denoted as o) caused by the slot table and the list data structure other than the data itself, as well as **extra overhead** (denoted as f) because of possible inaccurate estimations of the record size and memory page fragmentation. Here we define the fudge factor as $F = o * f$. Past literature has set the fudge factor to 1.2, but it

is not clear whether they have considered both kinds of overhead. In our experiments, the hash table overhead can be precisely computed based on the slot table structure; since we are using a linked-list-based table structure, there are 8 bytes of overhead for each slot table entry and 8 bytes of cost for each group in the list storage area. For the extra overhead, we tried four different ratios: 1.0, 1.2, 1.4 and 1.6. Figure 2.20 shows the running times. We can see that clearly it is not wise to consider only the slot table overhead ($f = 1.0$) since the running times of the Dynamic Destaging and Shared Hashing algorithms increase in both memory configurations. This is because the smaller fudge factor causes an underestimated partition size P, and thus there are partitions that fail to be fit into the memory during the hybrid hash. From our experiments we also observed that using slightly larger f values (> 1.2) has no significant influence on performance.

2.6 Summary

In this chapter we have discussed our experiences when implementing efficient local aggregation algorithms for Big Data processing. We revisited the implementation details of six aggregation algorithms assuming a strictly bounded memory, and we explored their performance through precise cost models and extensive empirical experiments. Among the six aggregation algorithms, we proposed two new algorithm variants, the Hash-Sort algorithm and the Pre-Partitioning algorithm. In most cases, the four hybrid-hash algorithms were the preferred choice for better running time performance. The discussion in this paper guided our selection of the local aggregation algorithms in the recent release of AsterixDB [1]: the Pre-Partitioning algorithm for its tolerance on the estimation of the input grouping key cardinality, the Sort-based algorithm for its

good performance when aggregating sorted data, and the Hash-Sort algorithm for its tolerance for data skew. We hope that our experience can also help developers of other Big Data platforms to build the solid local aggregation fundamental. In AsterixDB, based on this work, we are now continuing our study of efficient aggregation implementations in a clustered environment, where more factors like per-machine workload balancing and network costs must be further considered.

3

Global Aggregation

Compared with the local aggregation problem we described before, a global aggregation assumes that the input data may be partitioned and stored on different nodes. As the data is not a disjointly partitioned based on the grouping keys, simply running local aggregation algorithms on each of these partitions is not enough; intermediate results need to be merged in order to compute the global aggregation result.

In this chapter we extend our local aggregation techniques to search for efficient global aggregation plans. Specifically, we consider the global aggregation in a shared-nothing architecture. A common characteristic of this architecture is that each node has its own computation resource for independent data processing tasks, and nodes are interconnected through a network for communicating their intermediate results. This architecture provides very good scalability and failure recoverability and has thus been widely adapted by popular big data systems like Hadoop[14], Dryad[60] and our parallel platform Hyracks[8].

To process an aggregation query in a shared-nothing cluster, the most common approach is a "map-and-reduce" approach, where in the "map" phase input data parti-

tions are processed in parallel on the nodes containing these partitions, while the in the "reduce" phase the results of the map phase will be redistributed and merged to create the final global aggregation result. In a real scenario there could be multiple levels of "reduce" phases so the whole aggregation structure resembles a tree-like structure: leaf nodes contain the original input data partitions, while root nodes will get the final global aggregation result.

Given a "group-by" query over several input data partitions, it is the job of the query optimizer to assign proper local aggregation algorithms to each working node in the aggregation tree structure, in order to reduce the total cost efficiently. However this task is *very challenging*. First, as we have seen from the previous chapter, there are multiple options for the local aggregation algorithms for each node, and each algorithm behaves differently based on the CPU and I/O costs and the output data property (whether the output is sorted). An efficient global aggregation plan should use the proper algorithm according to the input data characteristics. Second, for global aggregation there is another important cost factor, the network I/O cost. In this environment the physical aggregation structure is also important, since different physical structures will lead to different costs on these factors. Third, different hardware environments have very different cost characteristics; the usage of advanced hardware, like SSD, Infiniband, has provided very different hardware performance patterns, so an efficient global aggregation plan in one environment may not be efficient anymore in another environment. It is thus important to generically model the cost of a global aggregation plan in a hardware independent way.

As a result, an efficient global aggregation plan is not a simple combination of the best local aggregation on each participating node. An efficient global aggregation strategy depends on both the input data statistics and the hardware characteristics.

The new network I/O cost introduces a new tradeoff between the effort for local volume reduction and the global data transferring cost, so picking the best local aggregation algorithm may not necessarily lead to the optimal global aggregation strategy. For example, depending on the input data statistics and the requirement of the output data, it could be beneficial to choose a local algorithm with higher cost but maintaining an interesting property like sorted order, or to do the "partial" local aggregation without completely aggregating the partition to avoid the high local cost for low data volume reduction (also known as "aggregation collapsing ratio").

In the following sections of the chapter we will address these challenges through a cost-model based approach. Our main contributions include:

- We extend our local aggregation study to include the optimization of partial local aggregation, where the local data may not be completely aggregated and the intermediate result will be merged globally for the final aggregation result.
- We discuss a hardware-independent cost model for global aggregation plans in a shared-nothing environment, considering the CPU, disk I/O and network I/O. We extended our local aggregation cost model to the global aggregation scenario to precisely model the cost involving partial aggregation and network I/O.
- We propose a cost-model based algorithm to generate the non-dominated global aggregation plan. A non-dominated global aggregation plan is a plan that will not be worse than any other possible global aggregation plan. This algorithm also supports user-defined weights on the cost factors, so that users can pick different preferences over the existing cost factors.

The remaining of the chapter is organized as follows: Section 3.1 reviews the literature about global aggregation. Section 3.2 provides the details about the global

aggregation plan structure, and the hardware independent cost model. Section 3.3 summarizes the local aggregation algorithms and also the connects (intermediate data redistribution strategies) that we will cover in this study. In the same section we also show the implementation details on these local aggregations as software components to maximize their reuse. We then describe a cost-model based plan searching algorithm in Section 3.4 to find the global aggregation plans with non-dominated costs, which supports weighted cost factors so it can be easily adapted to different hardware environment. Finally, Section 3.5 presents the experimental evaluation results for the global cost models and our proposed algorithm. Conclusions for this chapter appear in Section 3.7.

3.1 Related Work

The basic map-combine-reduce approach for global aggregation has been discussed in the NoSQL community [15], [43]; however, in this thesis, we also allow for the limited memory budget as well as the choice of different local aggregation algorithms with different connectors and partial aggregation choices. Global aggregation was also considered in the relational database research [50, 51],where several optimization strategies for global aggregation plans, including the different global aggregation network layouts and an adaptive local algorithm (hash-based) were presented no local aggregation options were provided, neither the different input and output properties were considered. In this thesis we also cover the hash-based local algorithm option (although without the adaptivity), but in addition, other local aggregation options as well as various input and output data properties are considered.

Aggregation trees have been discussed under many different scenarios like aggregating data streams in a sensor network [36] and iterative parallel processing [54]

[46]. In [46] an aggregation tree with different assumptions on the structure is discussed in the context of parallel multiple-level aggregation. The main difference is that the tree structure in this thesis assumes a fully hash partition between levels (while in [46] each node sends to a single node for merging the data), and further, we consider different local aggregation strategies.

[37] discussed a strategy similar to the PrePartitioning algorithm. The main difference is that they did not use any optimization to reduce the hash miss cost, but instead they used a dynamic strategy to spill the groups that have a low "absorb" ratio.

[27] studied the partial aggregation effects based on the clusterness of the input data from both a theoretical perspective and implementation. In our study we assume a random input dataset whose groups are uniformly distributed among the total input records, so that in our cost model we simply use cardinality ratio and the memory size to predict the partial aggregation effects.

3.2 Global Aggregation Model

Given a set of input data partitions and a set of nodes, the space of possible global aggregation plans is very large, due to the multiple choices on the local aggregation algorithms and also the different network topologies. In this section we describe our global aggregation model, which captures a subset of this plan space but contains the set of well-adapted global aggregation strategies used in the popular big data systems. We first describe the physical structure for a global aggregation implementation, called aggregation tree structure. Then we show the strategy that a query optimizer can use to map a logical aggregation plan onto a physical aggregation tree structure. Finally we discuss our hardware independent cost model framework for the aggregation tree

structure.

3.2.1 Physical Aggregation Plan: Aggregation Tree

The physical structure of a global aggregation plan describes how to parallelize the aggregation work over multiple nodes in a cluster environment. As our study of global aggregation assumes that the aggregation is processed over a shared-nothing cluster, the basic component of such a structure will be the standalone working node. Since each node has its own computation resources, including CPU and storage, the only way to share data between nodes is through the network connection between them. We do not consider the case for multiple cores or multiple threads running on a single node in our model described here, but the model can be extended to handle these cases as far as there are no sharing resources between different threads.

In our study we focus on a specific DAG structure for global aggregation, called *aggregation tree*. This structure divides the available working nodes into levels. Nodes within the same level will not have data communication, while nodes in two adjacent levels can send data from the lower level to the higher level (this is called a *data redistribution*). The leaf level nodes contain the original input data partitions, and the top-level nodes must produce the final aggregation results. Figure 3.1 thaws an example aggregation tree structure consisting of 9 nodes, among which 4 leaf level nodes contain the original input partitions. Note that nodes in the middle levels can also produce final results; this will be discussed in detail later in the specific local aggregation algorithms. We assume that the original input data in the leaf level nodes are randomly partitioned. Range partitioned and hash partitioned cases are not discussed in our study, as in both cases the final aggregation can be done by running a local aggregation to completely aggregate the records on each node, requiring no extra network traffic.

Specifically for the aggregation operation, each node in such a tree structure works as a pipe. Data is streamed into a node through the network from lower level nodes. Then the input data is processed internally in the node using some local aggregation algorithm. Note that the data volume could be reduced after the processing due to the aggregation operation. The result of the aggregation will then be sent and redistributed through the network to the next upper level nodes, as illustrated in Figure 3.1

In this structure there could be multiple input data streams and multiple output data streams for a single node. The *connector* defines the way of redistributing the intermediate results, by partitioning the single output stream onto all receiving nodes, and merging the multiple input streams into a single stream. From the view of a node there is always just one single input stream and one single output stream. Between levels, we assume the data redistribution strategy is *global hash partition*. This means that each record from a lower level node will be hashed onto *one* of the higher-level nodes. Nodes at the same level use the same hash function, so that the records of the same group residing at different nodes will be sent to the same node for aggregation. Although other data redistribution exist (e.g. range partitioning, and grouped hash partition as described in [46]), in this thesis we focus on this redistribution strategy because this is the well-known default partition strategy for mainstream big data systems. For example, in Hadoop by default the output of the mappers will be globally hashed onto all reducers, which makes sure that the data with the same key can be merged on the reducer side. Another reason for picking this redistribution strategy is that a global hash partitioning can evenly redistribute the data from the sender level to the receiver level, so that even the intermediate result in the sender level is not load balanced, after the hash partitioning the load will be re-balanced over the receiver nodes.

In this thesis we consider both the flat tree structure (i.e., a tree with only

two levels) and the deep tree structure (i.e., trees with more than two levels). A deep tree structure could be beneficial if the number of available nodes is very high. In that case the memory cost for building the global hash partitioning connections will be non trivial (recall that each sender node will need as many output buffers as the number of receivers; similarly, a receiver node needs as many input buffers as the number of sender nodes). Using a deep tree could then effectively reduce memory buffer cost for the redistribution.

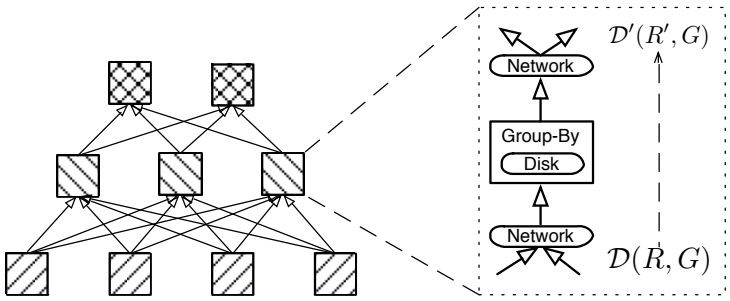


Figure 3.1: An 9-nodes aggregation tree (left) and the local node structure (right).

3.2.2 From Logical Aggregation Plan to Physical Aggregation Plan

Given an aggregation query with a set of input partitions and a set of available nodes, it is query optimizer's job to create an efficient physical parallel plan to utilize all available resources. Specifically, the query optimizer can assign different local aggregation algorithms to different nodes, in order to utilize the resource and minimize the overall cost. In our study, we discuss a common strategy for generating physical global aggregation plans in popular shared-nothing architectures, where a **logical sequential query plan** will be mapped into a shared-nothing parallel plan. A logical sequential plan consists of a series of local aggregation algorithms and the data redistribution strategies, which specifies the aggregation strategy for each level of the aggregation tree.

Then a query optimizer will map such a logical sequential plan into a physical logical plan, according to the data partitions. In details, the leaf level of the logical sequential plan will be mapped to the number of nodes that can handle all input data partitions. For the non-leaf levels, the query optimizer needs to decide how to arrange the nodes among these levels, in order to get the most cost-efficient physical plan. An example of this mapping from a 2-level logical sequential plan to a 5-node cluster is showed in Figure 3.2, where the leaf level of the logical sequential plan (using the disk-mode Sort-based algorithm discussed later) is mapped onto three nodes containing the input data partitions, while the root level is mapped to the remaining 2 nodes. Between the two levels nodes are connected using the HashMerge connection (discussed later) strategy in order to maintain the sorted order.

Based on this mapping strategy, for a physical aggregation plan, nodes on the same level run the same local aggregation algorithm, and also use the same hash redistribution strategy. Hence, nodes in the same level can be considered as "clones" of a single node. Such a physical plan structure can also be found in current big data systems. For example in Hadoop, the system will decide the copies of mappers based on the data locality, so that each node containing the input data will have at least one mapper instance cloned from the same mapper implementation. Then the number of reducers is decided according to the available system resources, by creating copies of reducers, each following the same reducer implementation.

3.2.3 Non-Dominated Global Aggregation Plans

In our study, we focus on the following three cost factors:

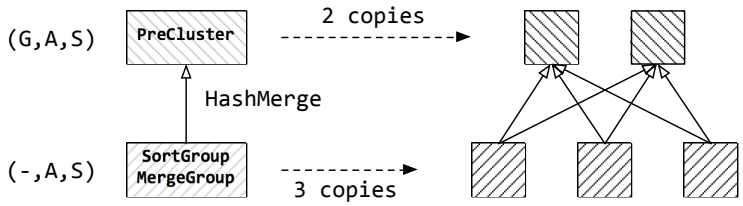


Figure 3.2: An example of properties, and the mapping from a sequential (logical) plan to a global (physical) plan.

- **CPU cost** denotes the sum of CPU cost from all nodes involved in the aggregation plan. Similar to the local aggregation study, CPU cost includes the comparison cost during sorting and hashing.

- **Disk I/O cost** denotes the total disk I/O on all nodes during the local aggregation. Many local aggregation algorithms, like the three algorithms discussed in the local aggregation work (Sort-based, Hash-Sort, and Hash-based), require disk I/O to completely aggregate the local records when the data cannot fit into memory.

- **Network I/O cost** denotes the total network transfer cost between levels during the aggregation. This is a new cost factor compared with the local aggregation. Basically network transfer is necessary as data of the same group but in different nodes must be sent to the same node through the network for merging.

The cost of a global plan will be a vector of these three costs of all the nodes and connections in the plan. The right part in Figure 3.1 illustrates the places where these costs could happen. For a local node, the network cost happens when the data is read in through the network, and when the result is sent through the network to the next level. The CPU and the disk cost occur during the local aggregation processing. Later we will show that the CPU cost will also appear when the input data needs to be merged to maintain the sorted order through the network transfer.

Note that all these three cost factors are hardware-independent, i.e. the same global aggregation plan will get the same cost vectors of these three factors when running on different clusters with different hardware configurations. By estimating these three cost factors through our cost model, specific hardware environments can apply different weights to these factors as the unit cost of these operations. For example, a magnetic disk based cluster could have a higher per-disk-I/O cost compared with a SSD based cluster.

Another important factor for searching the optimal global aggregation plans refers to the **output properties**. There are aggregation algorithms that could produce extra properties for the output. For example, Sort-based algorithm could produce sorted aggregation results. Such "properties of interest" could be utilized by the query optimizer if the properties are useful in the overall query plan, so plans generating results with different properties should not be compared directly for dominance. Furthermore, the output properties introduce the constraints on the combination of local aggregation algorithms and the connectors. Basically a connector that could destroy the "properties of interest" of the output of a node should not be used as the output connector for the node. We will discuss more about this in Section 3.3.

Formally we can define the problem of searching the non-dominated global aggregation physical plans as:

Definition 1 (Non-Dominated Global Aggregation Plan Searching Problem) *For an N-node shared-nothing cluster, given an input dataset $\mathcal{D}(n, m)$ (i.e., a dataset with total number of records n and containing m unique grouping keys) partitioned over $N_I \subset N$ nodes, this problem searches for the physical plans with the same output properties, so that each plan has at least one of the three cost factors (C_C, C_D and C_N)*

having cost that is no worse than any other possible physical plans.

3.3 Components for Global Aggregation Plans

In this section we discuss the components that could be used in a global aggregation plan. There are two main components for the global aggregation plans: **connectors** and **local aggregation algorithms**. Local aggregation algorithms determine the cost behavior of each local node, and connectors describe the data redistribution strategy between two levels. An efficient global plan should properly choose these components for each level of the aggregation tree, based on the system resources and the input data statistics.

Except for the cost, another constraint for the choice of the local aggregations and the connectors is the **output data property**. There are local algorithms that will introduce special properties to its output, and such properties could be useful for the remaining aggregation process, and even the operator after the aggregation in a complex query plan. So it is important that the connector and also the downstream local aggregations can utilize these properties properly, or just maintain such properties for further operations.

Based on these observations, in our discussion of these components, we will concentrate on the following aspects:

- Input Property Constraint: What is the requirement (if any) on the input data property for this component?
- Cost Behavior: What are the costs for all the three cost factors for a given input dataset?

- Output Property Constraint: What is the output data property (if any) of the output of this component?

For the cost behavior discussion, we use the symbols listed in Table 3.1; to save space, we omit the details of some cost formulas that have been discussed in the Cost Models section in Chapter 2.

Symbol	Description
b	Tuple size in bytes
o	Hash table space overhead factor (for its slot table and references of linked list)
p	Frame size in bytes
A	Collection of sorted run files generated
$\mathcal{D}(n, m)$	Dataset with n records and m unique keys
G	Output dataset size in frames
G_t	Number of tuples in output dataset
H	Number of slots in hash table
K	Hash table capacity in number of unique groups
N	Total number of nodes to be used in the global aggregation plan
N_i	Number of nodes in the i-th level of the aggregation tree
M	Memory capacity in frames
R	Input dataset size in frames
R_i	Input dataset size in frames for node i in the aggregation tree
R_t	Number of tuples in input dataset
R_H	Number of raw records inserted into a hash table before it becomes full

Table 3.1: Symbols used in Cost behavior discussion.

In the following subsections, we first discuss the possible output properties. Then we describe the connectors for different data redistribution strategies and the local aggregation algorithms.

3.3.1 Data Properties

Data properties are the properties that could potentially benefit the future query processing. In an aggregation tree, a property applies to the data partitions over

all nodes in the same level. In our current study, we are interested in the following three properties:

- Global Hash Partitioned: This property indicates that records of the same group will be on the same node in this level. This implies that the data are full hash partitioned over all nodes at the current level.
- Aggregated: This property indicates that the data on each node are completely aggregated (so on each node there is at most one records for a given group).
- Sorted: This property implies that on each node the data is sorted on the grouping condition.

Formally the data properties on a set of data partitions can be represented as a triple (G, A, S) (where G stands for global hash partitioned, A for aggregated and S for sorted). We use $(-)$ if the corresponding property does not apply for a set of data partitions.

Properties can be used to check whether a plan has reached the final result and can be terminated. Clearly, if the output of a level is both global hash partitioned and also aggregated, this output contains the final aggregation results. Hence, the query optimizer can always use the property triple $(G, A, -)$ to check whether the final aggregation result has been reached. Properties can also be used to constraint the choice of the operators and connectors. For example in Figure 3.2, since the output of the leaf level has the "sorted" property, it constraints the query optimizer to only choose the data redistribution strategy that could maintain the sorted order.

3.3.2 Connectors

A connector in the physical global aggregation plan specifies the data redistribution strategy from a set of sending nodes to a set of receiving nodes. A connector is defined by **a node map**, **a partition strategy** and **a merge strategy**. The node map is a map between the sending nodes and the receiving nodes and specifies the connections through which a sending node can send data to a receiving node. Each sending node can send data to multiple receiving nodes, and each receiving node could get data from multiple sending nodes. Formally such a node map can be represented using a $n \times m$ bitmap for n sending nodes and m receiving nodes. The partition strategy specifies the way how the data is partitioned if a sending node will send its data to multiple receiving nodes (like random partition, range partition, hash partition, etc.) The merge strategy describes how a receiving node would merge the multiple incoming data streams from multiple sending nodes into a single data stream. When the input streams are already sorted, a sort-merge strategy could be used to produce a single sorted stream.

For our implementation, in a connector, each sender will have an output buffer for each receiver it will send data to, and each receiver will have an input buffer for each sender it will receive data from. The partition strategy describes how the output records of a sender are partitioned onto the output buffers, and the merging strategy describes how the input records of the input buffers are merged.

Since in this study we assume a full hash partition redistribution strategy between two levels of nodes, the node map for any connector has a connection from each sending node to all receiving nodes. If the node map is represented using a bitmap, then all bits in the bitmap will be set to 1 (true). The hash partition strategy will also be the default partition strategy for all connectors. In the rest, we focus on the

following two different connectors with different merge strategies, namely the *Hash* and the *HashMerge* connectors.

- A **Hash Connector** has an on-demand, round-robin merge strategy. This strategy scans the input buffers in a round-robin fashion, and loads the first full buffer during the scan; it then releases the buffer to be filled up again by the input stream. This strategy has a non-deterministic data loading behavior over all the sending nodes.

- A **HashMerge Connector** uses a sorted merge for its merge strategy. This strategy waits until all the input buffers are filled with data. Then it starts a sorted merge to produce a sorted stream to the receiving node.

For the input data property, the Hash connector does not need any constraints over the input data properties. The HashMerge connector requires the Sorted property for the input data, so that it can guarantee the sorted order of its output data. This means that the local aggregation algorithm of the sending nodes should provide the sorted order on the output. For the output data property, clearly both connectors have their data globally hash partitioned, due to the full hash partition strategy. The hash merge connector further provides the Sorted property, so the local aggregation algorithm of the receiving nodes should be able to utilize the sorted order for efficient processing.

Although for a given input data set the network I/O for these two connectors is the same, their cost behavior differs. The main difference is that the HashMerge connector needs extra CPU cost on merging the sorted input streams using merge sort. For a connector from N_i sending nodes to N_{i+1} receiving nodes, if totally R_t is the total number of records that need to be sent through the connector, the total CPU comparison cost for merging is estimated as $C_{comp} = R_t * log_2 N_i$.

An important note about the implementation of these two connectors in a global aggregation plan is about the hash functions. Since there could be multiple levels in an aggregation tree, it is important to have different hash functions for the partitioning in different levels. Using the same hash function in the connectors of different levels will cause serious data skew during the redistribution, and thus downgrade the load balancing and eventually the aggregation performance.

3.3.3 Local Aggregation Algorithms

The local aggregation study showed that different local aggregation algorithms have varying cost characteristics. All three local algorithms are **blocking** in nature: in order to completely aggregate all local data, if the unique groups cannot fit into memory, intermediate data must be spilled and reloaded. However in a global aggregation plan, the local aggregation on a node may not need to completely aggregate its input data. Instead, the overflowing data can be directly streamed through the network to the next level of aggregation nodes. Depending on the trade-off between network cost and disk I/O, this strategy could potentially save disk I/O and utilize pipeline parallelism if the number of unique groups is large and not much local aggregation can be achieved to reduce the network cost.

Hence, for the global study we further extended the three local aggregation algorithms with fine-grained local aggregation operations that can choose to be in either the *disk-mode*, when it dumps the overflowing data onto disk (then reload it to continue the local aggregation for less network cost), or the *partial-mode*, when it streams the overflowing data directly through the network to the next level of the aggregation tree (thus avoiding disk I/O at the expense of increasing the network cost with incompletely aggregated data). The details of these improved local aggregation algorithms

are discussed below.

3.3.3.1 Improved Sort-Based Algorithm

When the memory is full and show be flushed, after sorting the data in memory, the improved Sort-Based algorithm will do a running aggregation over the sorted data, and then choose to send it through either network I/O or disk I/O. An special case is when the input data can fit in memory: then the data is directly sent through the network. Figure 3.3 illustrates this workflow. Specifically in the implementation, the flushing option is controlled by a flag "isPartial". The value of this flag is decided during the query optimization.

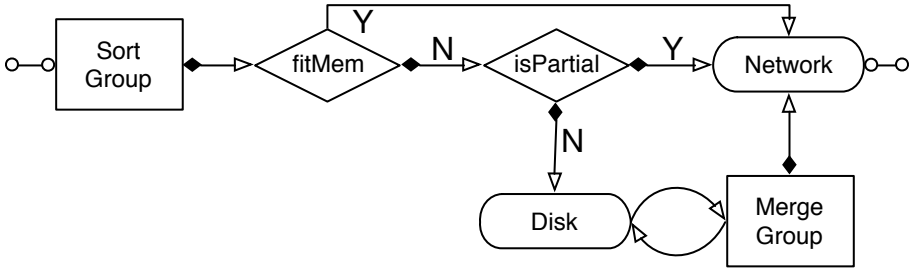


Figure 3.3: The Sort-Based Aggregation Algorithm Workflow.

Compared with the original Sort-based algorithm, this improved version introduces the optimization to pre-aggregate during the flushing, which reduces the I/O cost to either transfer the data through network or flush the data onto disk. In contrast, the running aggregation during the sort and merge phase introduces more CPU cost.

Both partial-mode and disk-mode of the improved Sort-based algorithm have no constraint on the input data property. The partial-mode algorithm produces results without any interesting property, as it only sorts and aggregates each full memory chunk. The disk-mode algorithm produces the results with aggregated and sorted property or

formally $(-, A, S)$.

The cost model of the improved Sort-Based algorithm includes the extra CPU cost for the running the aggregation during the sorting; this corresponds to R_{mem}, the number of raw records that can fit into the memory. Hence:

$$C_{comp} = |A| * (C_{sort}(R_{mem}, I_{key}(R_{mem}, R_t, G_t)) + R_{mem}) \tag{3.1}$$

If then the algorithm chooses to flush the memory through network I/O (i.e., partial-mode), there will be no extra disk I/O cost or merge CPU cost. The new network I/O cost will be the cost to send the $|A|$ sorted runs (containing only unique keys) through the network, namely:

$$C_{network} = |A| * I_{key}(R_{mem}, R_t, G_t)) * b/p \tag{3.2}$$

If the algorithm is in its disk-mode (i.e., data is dumped into sorted runs on the disk and the merge phase is required), during the merging phase, the number of records to be merged will also be reduced to contain only keys in each run.

$$C_{merge.comp} = C_{CPU.merge}(A, I_{key}(R_{mem}, R_t, G_t)) \tag{3.3}$$

Note that for implementation, the merge phase can be separated into a standalone component, since it can be shared by the improved hash-sort algorithm we will discuss below. The only difference for the merge phase between these two algorithms

is the key: for a sort-based algorithm, the sorting key is the same as the grouping key, while the hash-sort algorithm uses a combined key of (`hash-id`, `keys`).

3.3.3.2 Improved Hash-Sort Algorithm

The original Hash-Sort algorithm starts flushing when the in-memory hash table is full, and the groups in memory are first sorted based on the (`hash-id`, `keys`). In the improved version, at this point the algorithm can choose to either flush the result to the disk as described (i.e., disk-mode), or directly send the result through the network without first sorting it (i.e., partial-mode). Here the sorted order is not important when flushing through the network, because originally the sorted order is needed for the merge phase that follows it. If instead, the algorithm chooses to flush onto network, the global redistribution will not maintain the order anymore, so when the data reaches the next level, sorted order cannot be utilized based on the connectors we utilize in this thesis. Figure 3.4 illustrates the workflow of this improved version. As the improved Sort-Based algorithm showed in Figure 3.3; again the "isPartial" switch is used to control the data flow between the network and the disk.

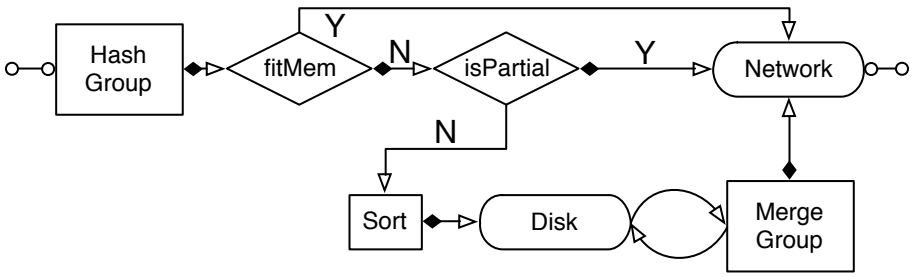


Figure 3.4: The Hash-Sort-Hybrid Aggregation Algorithm Workflow.

The improved Hash-Sort algorithm has no require on its input data property. For the output data property, the disk-mode produces aggregated data. Note that

although the disk-mode operator generates data sorted on the combined key (`hash-id`, `group-key`), the result is not total sorted on the grouping keys, so it does not have the sorted property.

The cost of the improved Hash-Sort Algorithm is different from the original version only in the partial mode, i.e. when the algorithm chooses to flush through network. In this case, the CPU cost contains only the hash aggregation cost:

$$C_{comp} = \frac{R_t}{|R_H|} * C_{hash}(|R_H|, G_t, K, H) \tag{3.4}$$

In its partial-mode, the algorithm produces no disk I/O. Instead all the results will be directly flushed to the network, introducing the new network I/O as:

$$C_{network} = \frac{R_t}{|R_H|} * \frac{Kb}{p} \tag{3.5}$$

where R_H is the number of raw records that can be hash-aggregated in the in-memory table before the memory is full, and K is the number of unique keys that can fit into the in-memory hash table.

3.3.3.3 Improved Hash-Based Algorithm

The improved Hash-based algorithm is based on the Pre-Partitioning algorithm discussed in previous chapter. Recall that the original algorithm divides the memory into two parts: one for an in-memory hash table to completely aggregate the resident partition, while the rest of the available memory will be used as the flush buffers for

the spilling partitions. Spilled partitions are processed in a recursive fashion. For the improved version, the algorithm can choose to flush the spilled records directly to the network in its partial-mode. However in this case it is unnecessary to create multiple spilled partitions. The reason is that since the data will be flushed to the network through full hash partitioned connectors, the spilled records will be re-partitioned by the connectors anyway. Based on this observation, in the partial-mode of this algorithm it always does Pre-Partitioning hybrid-hash with only one spilling buffer (so there is only one spilled partition), and utilizes the rest of the memory space for the resident partition. Moreover, there is no check on check whether a grace hash partition is needed. Recall that a grace hash partition is needed only if the input data is too large to use hybrid-hash algorithm, however in the partial-mode it does not need to guarantee that the spilled partition can be loaded back into memory for in-memory aggregation.

Figure 3.5 depicts the decision flow for this improved Hash-based algorithm. If the algorithm chooses to flush onto disk (i.e., disk-mode), it needs to check whether the grace partition is needed, and also it performs a fallback using a disk-mode Hash-Sort algorithm if the recursion level is too deep. If the algorithm chooses to flush onto the network (i.e., partial-mode), data will be partitioned into the resident partition (for aggregation) or the spilling partition (for spilling through the network).

When the improved Hash-based algorithm runs in the partial aggregation mode, and its input data has been globally hash partitioned, the resident partition will get the final aggregation results after all the input records are processed. In this case there is no need to send the resident partition through the network (which will increase the network cost and also the processing cost on the remaining levels), even if the current node is not a root node of the aggregation tree. Because of this scenario in

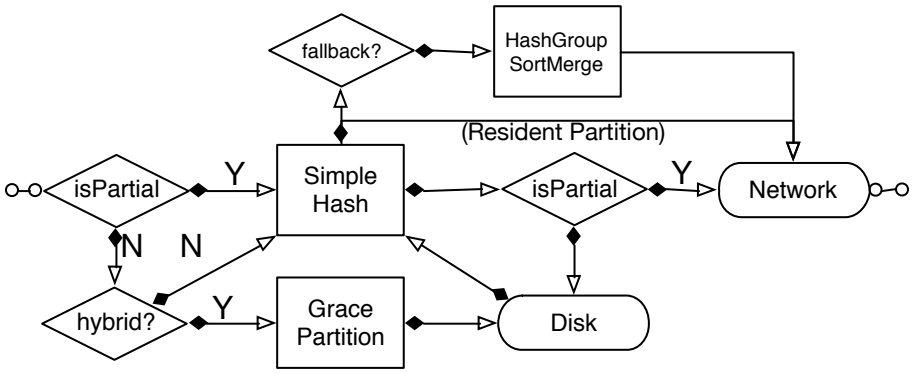


Figure 3.5: The Improved Hash-Based Aggregation Algorithm Workflow.

our aggregation tree model we allow non-root nodes of an aggregation tree to produce the final results directly without flushing through the network to the next level.

No specific input data property is required for this algorithm. And its disk-mode will produce aggregated results, but no other interesting properties. For the cost model of the partial mode, the CPU cost consists of the cost for aggregating the resident partition records, and the cost to identify the spilling records after the hash table is full. Since there is only one spilling buffer in memory, the number of resident groups is $G_{res} = (M/F * p/b)$ (the fudge factor is applied here to take the overhead cost into consideration). For the hash miss cost for the spilling partition we again use the bloom-filter based hash table with false-positive rate of α to reduce the hash miss:

$$\begin{aligned} C_{comp} =& C_{hash}(\frac{R_t * G_{res}}{G_t}, G_{res}, G_{res}, H) \\ &+ \alpha(R_t * (1 - \frac{R_t * G_{res}}{G_t}) * \frac{K}{H}) \end{aligned} \tag{3.6}$$

The new network I/O cost for flushing the spilling partition is:

$$C_{comp} = R * (1 - \frac{R_t * G_{res}}{G_t}) \tag{3.7}$$

3.3.3.4 PreCluster Algorithm

Here we introduce *PreCluster*, a new local aggregation algorithm utilizing the sorted intermediate results. This algorithm assumes that the input data is already sorted (i.e., with sorted property for input data) on the grouping key and it guarantees that is output is also sorted, by simply performing a running aggregation on the input data streams. The CPU cost of this algorithm is simply $C_{comp} = R_t$, as the running aggregation does only comparisons for each input record and produces no disk I/O cost. The network cost, if applicable, will be the cost to send the unique keys or formally $C_{network} = G_t$.

3.4 A Cost Model Based Algorithm for Non-dominated Global Aggregation Plans

3.4.1 Rules for Efficient Global Aggregation Plans

So far we have discussed the components of building a global aggregation plan, including the network structure of a plan, connectors and local aggregation algorithms. Combining these components together leads to a huge number of possible aggregation plans. However, not all of these plans are of interest. To generate the efficient plans, we propose rules (when building the aggregation tree) to eliminate inferior plans. In particular:

- Nodes used in an aggregation plan should perform some non-trivial data processing

task. (That is, we want to avoid plans where nodes only pass their input data to the output stream without doing any aggregation processing).

- Constraints from the data properties should be enforced when building the aggregation tree, so that interesting properties can be utilized, and will not be destroyed by the downstream operations.

The first rule will eliminate an aggregation tree having non-root levels with only one node. Typically, this one-node level simply passes the input data onto the next level without doing any meaningful aggregation work. This is because most of our local algorithms the in-memory aggregation collapses the duplicates in each page sending to the single node of the next level. For the local algorithms that will send pages with duplicates, specifically the partial-mode Hash-based algorithm, the spilled partitions will be sent to the single node level for aggregation. An extreme example would be a chain of improved Hash-based nodes. We also prune this structure because it is worse than a "fat tree" structure where the processing of the spilled partitions can be paralleled among nodes. (Basically a deep tree utilizes the pipeline parallelism, while a fat tree utilizes the partition parallelism.)

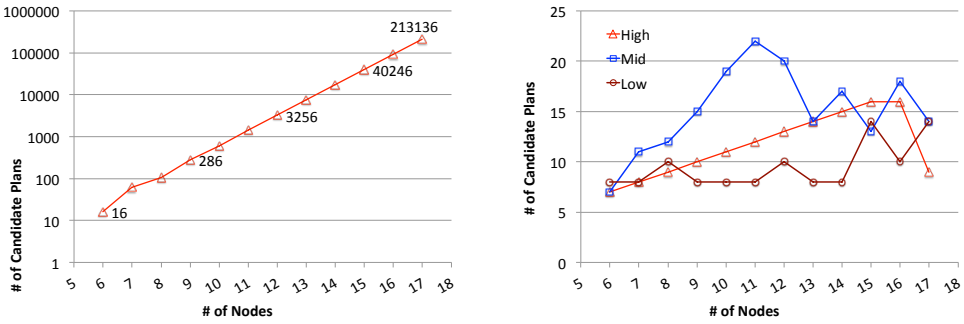
Furthermore, an aggregation tree with nodes having disk-mode local aggregation algorithms at the middle levels (i.e., not the leaf level or the root level) can also be eliminated by the first rule. This is because our disk-model local aggregation algorithms can completely aggregate the input records. Since the data streamed to such a middle level node definitely has the property of globally hash partitioned, the aggregation result of such a node will be the final result and need not to be processed again.

The second rule will enforce the sorted order to be properly utilized and maintained. Specifically we have:

- Nodes in a level with disk-mode improved Sort-based algorithm must have a Hash-Merge connector for their output.

- A HashMerge connector must send data to nodes with the PreCluster algorithm in order to utilize the sorted order.

3.4.2 Cost Model Based Plan Searching Algorithm



(*a*)All Candidate Plans

(*b*)Non-Dominated Plans for Different Cardinalities.

Figure 3.6: Number of candidate plans (a) and non-dominated plans (b) for different nodes.

Unfortunately simply applying the rules discussed above will not eliminate many inefficient candidate plans. Figure 3.6 shows the number of generated candidate plans after considering the above two rules in a log-scale chart, for clusters from 8 nodes to 17 nodes, where the input data partitions are stored in 4 nodes. Clearly, there is an exponential increase for the number of plans.

To pick only the efficient plans, we propose the following algorithm that finds the *non-dominated* plans, utilizing our cost models for the aggregation tree structures. The pseudocode is shown in Algorithm 1. Using a dynamic programming approach, we recursively search the non-dominated sub plans for a subset of available nodes, and group the non-dominated sub plan candidates based on their output properties. During

the search, sub plans using a specific number of nodes will be stored (to be used later when sub plans with this number of nodes are requested). To speedup the search, we also enable plan pruning, so that only non-dominated plans will be stored.

This algorithm can greatly reduce the number of plans to be considered for efficient global aggregation. Figure 3.6 (b) shows the number of non-dominated plans for the same cluster configurations as in Figure 3.6 (a) for three different data cardinalities: high cardinality ($> 90\%$ unique groups), median cardinality (around 30% unique groups) and low cardinality ($< 1\%$ unique groups). Although the number of candidate plans increases exponentially with the number of nodes, the number of non-dominated plans remains small (around 15).

Algorithm 1 Search-Plan(N, N_I)

Require: L as a hash map of mapping candidate plans to the key as the number of nodes used; N as all the available nodes; $N_I \subset N$ as the nodes containing the input data partitions; and P_D as the property of the data.

```
1: if L[N] is not NULL then
2:    RETURN L[N]
3: end if
4: create a hash map H_N of (property, plans) pairs, and insert it into L with key N
5: if N == N_I then
6:    generate the leaf level: for each local algorithms satisfying the constraints from
      property P_D, and generate the sub plans. Insert the non-dominated plans into
      H_N using the output property as the key.
7: else
8:    for i = (1, ..., N − N_I) do
9:       Search-Plan(L, N − i, N_I)
10:      Retrieve the sub plans using N − i nodes, as a hash map H_{N−i}
11:      for all connector C that satisfies property P'_D do
12:         for all local algorithm A_l satisfying property of output of connector C do
13:            generate sub plans by adding a new i-node level over the all sub plans
               in H_{N−i} using C and A_l, and insert the non-dominated sub plans in H_N
               according to their output property.
14:         end for
15:      end for
16:   end for
17: end if
```

This algorithm can be easily extended to include customized cost weights. De-

pending on the actual application, such a cost weight can be the average time for each unit of operations (CPU comparison, disk I/O or network I/O), or the energy consumption of each unit of operations. By applying customized cost weights the algorithm will return a single plan for each data property of interest. For accommodating cost weights, we only need to change the logic on checking the domination relation (Line 6 and Line 13 in Algorithm 1).

We note that since we are considering three cost factors, the problem of finding the non-dominated plan is actually a *skyline* problem. Several research works [57], [39] consider how to efficiently reduce the number of skyline objects, while guaranteeing that the remaining objects can represent the removed non-dominated plans.

3.5 Experimental Evaluation

We proceed with an experimental study on the global aggregation cost models and the cost model based plan searching algorithm. We ran our experiments on a single machine with Intel Core i7 2.2 GHz (4 cores with 8 threads) with 8GB 1333 MHz DDR3 main memory, and 750GB 5400 RPM hard drive. The software system is Mac OS X 10.9, and JDK 1.7.0_45. For these experiments we used 8 Hyracks instances running in a virtual cluster on the single test machine. Note that since our cost model is hardware independent, the results on the cost factors examined in this section will not be changed if the same experiments are moved to another environment (this would only affect the total running time).

We used the same aggregation query considered for the local aggregation study, but on a smaller uniform data set with total 10 million records (218 MB in size). Further, we considered two different group cardinalities, a low-cardinality with 15k groups and

a high-cardinality with 9.8 million groups. For this data set we created four partitions, each with 2.5 million records. Our aim is to run all possible aggregation plans so as to verify the model correctness, hence the reason for picking this small dataset. However note that our global aggregation model can be easily ported to a larger scale system for larger data sets.

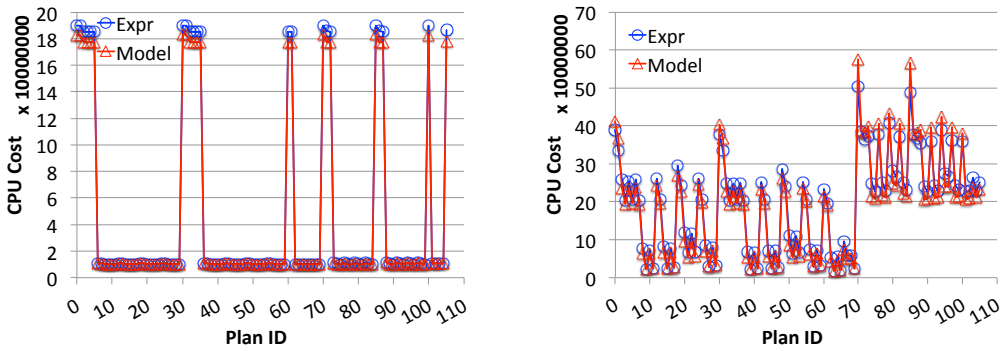
For the given configuration, using the two rules described in Section 3.4.1 (i.e., no dominance checking), there are totally 106 candidate plans created. Among these plans, there are two different output properties: 70 plans come with (G, A, -) (i.e. the result is fully hash partitioned and aggregated) and 36 plans with (G, A, S) (i.e. the result is fully hash partitioned, aggregated, and also sorted). Among the 70 unsorted plans, there are 30 plans whose nodes in aggregation tree are in the form of 4-3-1 (i.e., 4 nodes in the leaf level, 3 in the middle level and 1 root), 30 plans with nodes following a 4-2-2 structure, and 10 plans with a 4-4 structure. For the 36 sorted plans, there are 15 plans with a 4-3-1 structure, 15 plans with a 4-2-2 structure, and finally 6 plans with a 4-4 structure. While varying the local aggregation strategies for the same tree structure, we started from the root level to the leaf level, and used partial aggregation algorithms first and disk-mode algorithms next.

3.5.1 Cost Model Validation

To validate the global cost model, we compared the cost model prediction with the actual cost numbers from the real experiments. In order to cover both the in-memory aggregation cases (i.e. the group cardinality of the dataset is small enough to fit in memory) and the spilling cases (i.e., the group cardinality of the data set is larger than the memory, so spilling is required), in the experiments we used a small memory configuration (16MB), so that the low-cardinality dataset can be fully aggregated in

memory, while the high-cardinality dataset will spill. In order to compare all possible plans, we disabled the dominance check when generating the plans, so all valid candidates for the two output properties, including 70 plans for unsorted results and 36 plans for sorted results, are checked.

Figures 3.7, 3.8, 3.9 show the comparisons between the global cost models and the real cost numbers from for all the 106 plans generated in the experiments. Overall our cost model can precisely predict the actual costs for all three factors, for both the low-cardinality case and the high-cardinality case. For the CPU cost, from Figure 3.7 (a) we can see that since the input data can fit into memory, most of the plans need very limited CPU cost. There are plans using much higher CPU, because these plans use the improved Sort-based algorithm in some of its nodes. For the high-cardinality dataset in Figure 3.7 (b), the CPU cost varies a lot since in this case the high cardinality causes many CPU comparisons for Hash-Sort algorithms (very little collapsing but frequent sorting and flushing; also with high number of hash misses). In the same figure we observe a higher CPU cost on average from plan 46 to plan 76. This is because all these plans all have the Sorted output property, so their final levels always use the disk-mode improved Sort-based algorithm, which causes a high CPU comparison cost.

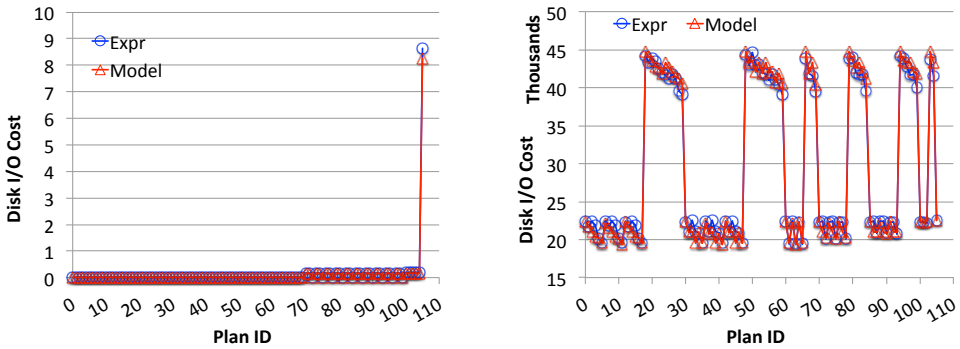


(*a*)Low Cardinality (*b*)High Cardinality

Figure 3.7: Global CPU I/O cost model validation.

Figure 3.8 shows the disk I/O cost for all the candidate plans. Again, the cost model predictions match with the real cost numbers very well. In the low cardinality case all the plans for unsorted results (plan 0 to plan 69) need no disk I/O at all. For plans generating the sorted results (71 through 105), the only local algorithm that can get the sorted order is the disk-mode Sort-based algorithm, which needs disk I/O for flushing and merging sorted runs. The extreme case is plan 105, which is a two level aggregation tree with the first level using the disk-mode Sort-based algorithm and the second level using the PreCluster algorithm. Since the disk-mode Sort-based algorithm has the least group collapsing effect, while it also needs flushing and merging I/O cost, it uses a very high disk I/O when compared with other plans.

For the high cardinality case showed in Figure 3.8 (b), the disk I/O rises regularly in the generated candidate plans. All plans having a spike in the disk I/O have at least one disk-mode local algorithm. For example, the plans 18 through 23 in the first level use the disk-mode Hash-Sort algorithm, while the first level in plans 24 through 29 uses the disk-mode Hash-Based algorithm. Due to the high cardinality, most of the records will be flushed onto the disk in order to be fully aggregated locally.

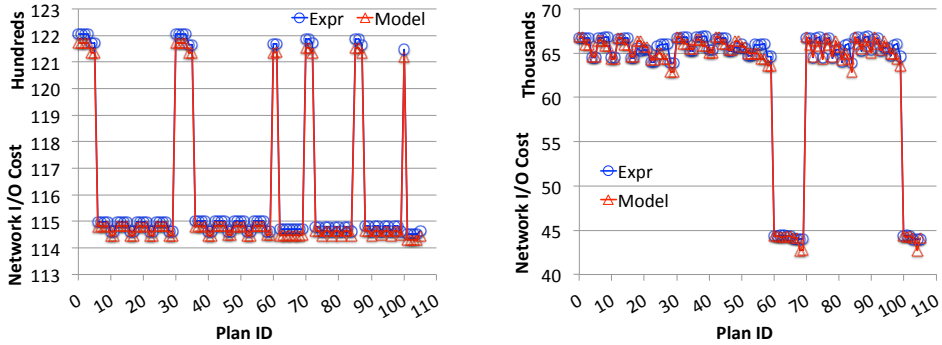


(*a*)Low Cardinality (*b*)High Cardinality

Figure 3.8: Global disk I/O cost model validation.

Figure 3.9 depicts the network I/O cost for the considered plans; the cost model

closely predictsthe network cost as well. In the low cardinality case, the collapsing effects of different local algorithms greatly affect the network I/O: since more records can be aggregated on the local node, less records need to be sent through the network. Among these algorithms, the partial Sort-based algorithm has the least collapsing effect as it can only aggregate a full memory of records each time. The spikes in Figure 3.9 (a) indicate the plans using partial Sort-based algorithms. On the contrary, the hash-based algorithms can collapse the data during the insertions, so more records can be aggregated, resulting in the low value regions between the spikes. The high-cardinality case depicts a very interesting pattern, where for the plans with a given output property, the last several plans always have a lower network I/O. This is because these plans only have 2 levels in their aggregation tree structures (each level with four nodes). In the high cardinality case, most of the records are unique so all unique groups will be transferred between levels; thus more tree levels will cause larger network cost.



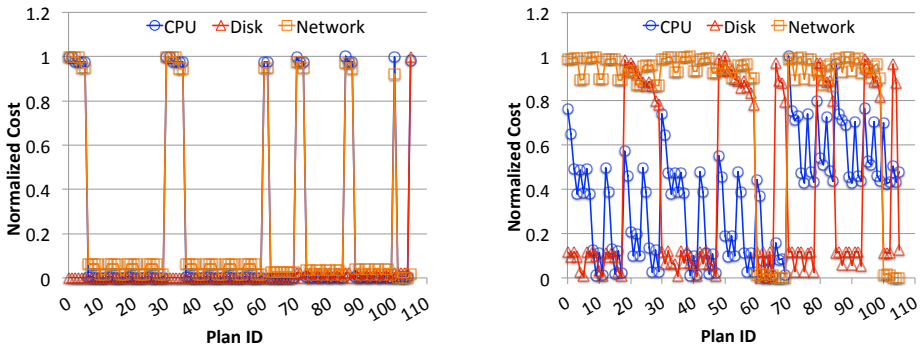
(*a*)Low Cardinality (*b*)High Cardinality

Figure 3.9: Global network I/O cost model validation.

3.5.2 CPU, Disk, Network Cost Interaction

Our experiments depicted interesting interactions among the three cost factors. Figure 3.10 draws the three cost factors into the same figure after normalization. In

the low cardinality case (Figure 3.10 (a)), the CPU cost and the network cost exhibit very similar behavior among most of the plans. This is because in the low cardinality case, higher CPU cases always indicate plans using the Sort-based algorithm, which has the least aggregation collapsing effect thus leading to a higher network cost. The only exception is plan 105, where the disk I/O is high but the network I/O is low. As mentioned above, this plan has only two levels with the first level using the disk-mode Sort-based algorithm; this implies high CPU cost for the Sort-based aggregation and also disk I/O for flushing and merging sorted runs.



(a)Low Cardinality (b)High Cardinality

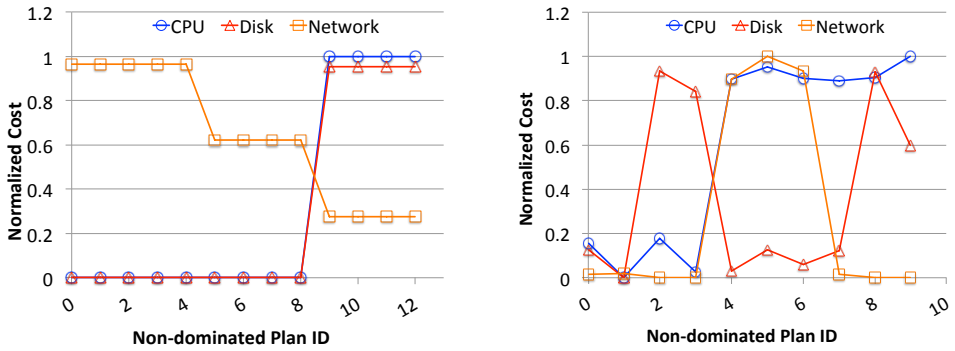
Figure 3.10: Normalized costs in the same chart.

In the high cardinality case (Figure 3.10 (b)), there is no clear pattern over all plans. Nevertheless, plans using the Sort-based algorithm incur both higher CPU and network I/O cost, since the sort operator is CPU expensive, and each time only the records in the memory can be aggregated. The usage of hash-based algorithms, especially the Hash-based algorithm can greatly reduce the CPU, disk and network cost. In the two-level plans (plans 60 through 69 and 100 through 105), there are cases where the disk I/O is very high (plans 64 through 69, and 102 through 105). This is because in a two-level aggregation tree, both levels can choose the disk-mode local algorithms. In the high cardinality case, not much collapsing can be achieved in the first

level, so the disk cost for the second level is also high. Such large disk I/Os cannot be found in the low cardinality case because the data can fit in memory without causing disk flushing.

3.5.3 Non-dominated Global Aggregated Plans

Here we ran our proposed search algorithm to find the non-dominated plans for both the low cardinality and the high cardinality cases. For our configuration using a total of 8 nodes with 4 nodes having the input partitions, there will be 13 non-dominated plans for the low cardinality case (5 for unsorted results, and 8 for sorted results) and 10 non-dominated plans for the high cardinality case (4 for unsorted results, and 6 for sorted results). Table 3.2 and Table 3.3 show the non-dominated plans found by our algorithm among the original set of 106 candidate plans. Since these plans achieve the Pareto optimization, it is not necessary for them to have the best cost on all three factors; instead some factors among these three are no worse than any other plans. This is also illustrated in Figure 3.11 that depicts the the normalized costs for all non-dominated plans.



(*a*)Low Cardinality (*b*)High Cardinality

Figure 3.11: The normalized cost for non-dominated plans.

For the low cardinality case, shown in Figure 3.11 (a), there are groups of

plans (like plans 0 through 4, 5 through 8 and 9 through 12) that have relatively, very similar behavior. For example, all plans from 0 to 4 appear to have the same three costs. This is because in low cardinality, different local algorithms typically aggregate the local data in main memory, resulting in similar behavior. The plan searching algorithm still provides all these plans as non-dominated because the actual costs are slightly different (but cannot be depicted in the figure).

PlanID	Level 0 (# nodes)	Level 1 (# nodes)	Level 2 (# nodes)
62	HashSort[P](4)	HashSort[D](4)	-
64	HashBased[P](4)	HashSort[D](4)	-
66	HashSort[D](4)	HashSort[D](4)	-
68	HashBased[D](4)	HashSort[D](4)	-
69	HashBased[D](4)	HashBased[D](4)	-
81	HashSort[D](4)	HashBased[P](3)	SortBased[D](1)
84	HashBased[D](4)	HashBased[P](3)	SortBased[D](1)
96	HashSort[D](4)	HashBased[P](2)	SortBased[D](2)
99	HashBased[D](4)	HashBased[P](2)	SortBased[D](2)
101	HashSort[P](4)	SortBased[D](4)	-
102	HashBased[P](4)	SortBased[D](4)	-
103	HashSort[D](4)	SortBased[D](4)	-
104	HashBased[D](4)	SortBased[D](4)	-

Table 3.2: List of non-dominated plans for low cardinality dataset.

The high cardinality case in Figure 3.11 (b) shows a variation of the cost factors for different non-dominated plans, however each plan cannot be dominated since at least one of its cost factors is better when compared with other non-dominated plans. In the unsorted plans, although the last two unsorted plans have high disk I/O cost but just slightly better network I/O, they are returned as non-dominated plans since in our experiments we treat all the three cost factors evenly. In reality when disk I/O is considered to be very expensive compared with the CPU and network I/O costs, these plans could be pruned by applying a higher weight to the disk I/O factor (which means that the cost of a unit disk I/O operation is more expensive). In the sorted plans, we

observed a similar trade-off between disk I/O and network I/O, where for the last two plans the high disk I/O is from the disk-mode local algorithms on the leaf level, and the low network I/O is because there are only two levels in these plans.

PlanID	Level 0 (# nodes)	Level 1 (# nodes)	Level 2 (# nodes)
62	HashSort[P](4)	HashSort[D](4)	-
63	HashSort[P](4)	HashBased[D](4)	-
68	HashBased[D](4)	HashSort[D](4)	-
69	HashBased[D](4)	HashBased[D](4)	-
75	HashSort[P](4)	HashBased[P](3)	SortBased[D](1)
89	HashSort[P](4)	HashSort[P](2)	SortBased[D](2)
90	HashSort[P](4)	HashBased[P](2)	SortBased[D](2)
101	HashSort[P](4)	SortBased[D](4)	-
104	HashBased[D](4)	SortBased[D](4)	-
105	SortBased[D](4)	PreCluster(4)	-

Table 3.3: List of non-dominated plans for high cardinality dataset.

3.6 Discussions

While the proposed plan searching algorithm is able to identify the relatively few non-dominated plans, its running time depends on the cost model computation for each plan examined. Among the different algorithms the hash-based cost model component is computationally expensive because we simulate the hash table look-up procedure. During the simulation we compute the hash cost for each insertion until the hash table is full; this depends on the input size and the hash table capacity. We suggest two approaches to deal with this issue. First, we could precompute this simulation for many common input and hash table parameters. Then, during the plan search, the algorithm could look up the precomputed cost. This of course implies an extra cost for storing these pre-computations and requires an estimation procedure in case the exact parameters have not been precomputed already. Another approach is to use a faster but

less precise cost model for hash-based algorithms. Both cases are interesting to pursue and we leave them for further research.

Note that the proposed algorithm follows a top-down recursion approach, so the subplans (i.e., plans not using all available nodes) are built before any full plans (i.e., plans using all available nodes). From our experiments we made two observations: (i) most non-dominated plans contain no more than three levels in their aggregation tree structure, and (ii) there are very few three level plans (most of the three level plans are pruned). This is because a higher aggregation tree will potentially increase the network cost, while it could bring limited benefits on the CPU and disk I/O cost.

Based on this observation, an improved plan searching algorithm can be proposed using a bottom-up strategy. Instead of building the incomplete subplans first, the algorithm can start to build the completed plans from the minimum number of levels (2 levels in our model). By iterating the possible numbers of nodes in the second level, both the completed plan (using all remaining nodes in the second level) and the incomplete 2-level subplans (there will be nodes left unused for more levels) can be built. Then the algorithm continues recursively to build higher trees based on these existing 2-level subplans. Each such subplan will be maintained in a seed subplan list, and once it is used to expand into higher subplans, it will be removed from the list and the new generated subplans will be added into the list. The algorithm terminates when the list is empty.

Note that if the no dominance relation is checked for the seed subplans, all possible subplans will be checked. To maximize the pruning over the subplans, the following two strategies can be used to prune the subplans that would be inserted into the list: (1) only the subplans that will not be dominated by any completed non-dominated plans will be inserted into the seed list, and (2) to handle the subplans

with no disk I/O cost (i.e. there is no disk-mode algorithms used in the subplan), the algorithm estimates their possible disk I/O using the best disk I/O from the completed non-dominated plans so far. Note that the second strategy could potentially prune the subplans that could lead to a lower disk I/O, however in practice we notice that this is rare, because the completed 2-level non-dominated plans always contain the plan with the best disk I/O, which has fully utilized the in-memory aggregation in all available nodes. This technique can be used to speed up the plan searching procedure, but not necessarily guaranteeing that all the original non-dominated plans will be returned.

Furthermore, we also observed the following behaviors on choosing the local algorithms for a global aggregation plan (under the assumption that we do global hash partitioning between levels):

- If the input data is already sorted, the PreCluster algorithm can be directly applied to completely aggregate the data without extra disk I/O cost.

- When the group cardinality of the input is low, it is beneficial to pick the partial-mode HybridHash-based local algorithm to achieve good aggregation collapsing. Partial-mode Hash-Sort algorithm is the same good if the data can fit in memory, but may have a lower aggregation collapsing effect if the data needs to spill. The partial-mode Sort-based algorithm would be good if the whole input dataset (not aggregated) can fit in memory.

- When the group cardinality of the input is high, the partial-mode Hash-Sort local algorithm is a good choice as it could aggregate the input records using hashing (so its CPU cost is low), while it avoids the high hash miss cost of the partial-mode HybridHash-based local algorithm. Partial-mode Sort-based algorithm is not recommended because it has high sorting cost, and also the latest aggregation

collapsing effect.

3.7 Summary

In this chapter we discussed global aggregation strategies in a shared-nothing environment. A global aggregation plan can be physically represented as an aggregation tree, where a tree node represents the local aggregation strategy and an edge indicates the data redistribution strategy. Since the aggregation result is generated by the root level nodes in an aggregation tree, it is also possible to use partial local aggregation algorithms. However given the different aggregation tree structures, different local aggregation algorithms and different data redistribution strategies there is an exponential number of physical aggregation plans. In order to identify the efficient global aggregation plans, we extended our local aggregation cost models to predict the cost of the global aggregation plans. Our cost model uses three hardware independent cost factors: CPU cost, disk I/O cost and network I/O cost. Through this cost model, we also proposed an algorithm to find the global aggregation plans with non-dominated costs. From our experiments we have showed that our extended global aggregation cost model can precisely predict the global aggregation plan cost, while our algorithm can find the typical small number of non-dominated aggregation plans without searching the whole candidate plan space.

4

Continuous Top-k Objects with Relational Group-By

4.1 Introduction

Nowadays most of big data applications need to handle the large volume datasets with spatial and temporal information enriched. Such data are very common in various areas, like the social network, log information, business transaction, etc. Most of the traditional data processing tasks, like join and aggregation, do not directly address these enriched information, so there have been a burst increase on research topics about handling spatial and temporal enriched big data. Specifically, traditional data processing tasks may be extended to cover the new requirement from these new information. Efficient algorithms could be devised based on the properties of these spatial and temporal information, which is not possible for the tradition data set without such information.

In this chapter we will discuss a new aggregation problem, focusing on the spatial and temporal information enriched dataset. Compared with the aggregation

problems we have discussed in previous chapters, the main difference of this new aggregation problem is that the grouping information is position and time related, and it is unknown before the the related data is collected and processed. In this new aggregation problem, the group of an object is subject to the other objects around it in the scope of both position and time.

Formally, we call this new aggregation **relational group-by**. The definition of such a relational group-by is an input dataset, an aggregate function, and a relational group function. For the aggregation function, it is the same as what we have discussed in the previous chapters, like SUM, COUNT, MIN, MAX, AVG, etc. The main difference here is the relational group function, defined as below:

Definition 2 *Relational Group Function: A relational group function, F_G, accepts the input of two data tuples r_0 and r_g, and returns a boolean value illustrating whether r_g will be involved into the score calculation of r_0. That is, $F_G : r_0 \times r \rightarrow BOOL$.*

Basically this relational group function defines the groups. For any two records, we can check whether they are in the same group or not by applying the relational group function over them. The traditional aggregation algorithm can be considered as a special case, where the relational group function simply checks whether the two records have the same grouping key value.

This chapter we will discuss this new aggregation problem in a spatial and temporal context. An example application for this specific problem is like this: For ensuring public safety, objects (or *protectees*) like cash transport vans and bank branches, should be properly protected by police forces (or *protectors*) patrolling around. Crimes can be prevented if all protectees are protected well and offenders cannot find any weak point to attack. However, since there is a limit on available protectors and some

objects are moving, it is important to recognize protectees without enough protection for potential attacks. Monitoring such unsafe objects is also important for the efficient arrangement of protectors, where additional protection force should be placed to protect unsafe objects instead of strengthening the safe ones.

In our framework, different protectees have different *safety requirements*, while different protectors can provide different amounts of protection (*safety supply*). Some protectees, like cash transport vans and vehicles of foreign dignitaries, need more protection than others. Different types of protectors, like police helicopters, motor cruisers, mounted officers, and walking patrols, have different protection capability and different size of protection regions. More importantly, both safety requirement and supply may change over time since some objects are moving, and at different time or places the safety requirement/supply varies (e.g. some places may have higher crime rate, the configuration of police force may change, etc.). In order to handle these dynamic changes, continuous monitoring is necessary on the most unsafe objects to be protected.

This chapter proposes a novel continuous top-k query, called *continuous top-k unsafe moving object query* or *CTUO*. Each object, either a protectee or a protector, is modeled as a record with spatial and temporal information and safety weight (safety requirement for protectees, safety supply for protectors). The *unsafety weight* of a protectee, which works as the score in the top-k query, is defined as the difference between its safety requirement and the protection it can obtain from protectors around it.

CTUO is a novel continuous top-k query on location-based data. In order to calculate the unsafety weight, both protectees and protectors should be considered in a "join-group-sort" procedure. Given a protectee and a protector, CTUO needs to decide whether the protection relationship exists. Then for all protectors protecting a given

protectee, an aggregate should be processed to obtain the unsafety weight. Finally all protectees are sorted on their unsafety weights and the top-k unsafe objects can be decided.

The computational challenge introduced by CTUO cannot be handled by existing methods. Since CTUO describes the significance of relationships between protectees and protectors, traditional top-k algorithms with scores as the characteristics of objects cannot be applied here. Two similar queries are *rank-join* and *rank-aggregate*. However, rank-join queries only consider the score of each join result individually, and rank-aggregate works on groups in a single relation. CTUO discusses the ranking query combining these two queries, which cannot be solved efficiently by solutions for either of them.

This chapter proposes two algorithms, *GridPrune and GridPrune-Pro*, to efficiently solve CTUO queries. Both these algorithms use the basic prune strategy, BoundPrune, which is also widely applied in many top-k algorithms. Although BoundPrune can be used for more generic CTUO queries where arbitrary join condition and any distributive aggregate are allowed, its performance is quite limited due to the loose pruning bounds. The two algorithms we propose improve this by utilizing a grid index to boost the prune power.

Our contributions in this chapter can be summarized as follows:

- We propose the continuous top-k unsafe moving object query or CTUO. This introduces a novel type of continuous top-k queries describing the significance of protection relationships between protectees and protectors. It is also important to many real applications to monitor public safety and arrange safety enforcement.

- We discuss the prune strategy using bounds of scores and a basic pruning algorithm

BoundPrune, which is a generic algorithm working for arbitrary join and score conditions defined in CTUO.

- We propose two algorithms by improving the prune strategy using a grid index structure: GridPrune for top-k unsafe object query on both static and continuous (efficient for batch updates), and GridPrune-Pro for continuous top-k unsafe object query. Our experiments show great I/O performance improvement from the proposed algorithms compared with the naive approach.

- We evaluate our proposed algorithms in both memory-based and disk-based implementations, and discuss the performance gains under different parameters.

The rest of the chapter is organized as follows. Section 4.2 reviews existing works related to CTUO, and Section 4.3 formally defines the CTUO. In Section 4.4 we introduce the prune strategy by bounding and pruning the unseen objects, and in Section 4.5 and 4.6 we present two efficient algorithms to solve CTUO queries, GridPrune and GridPrune-Pro. Section 4.7 describes the experimental evaluation while Section 4.8 concludes the chapter.

4.2 Related Work

Continuous Top-k Unsafe Place: This chapter extends and generalizes the *continuous top-k unsafe place query* or *CTUP* proposed in [62]. Given a set of places with different safety weights and moving protection forces, a CTUP query monitors the most k unsafe places over updates of the protection forces. Two I/O efficient algorithms, *BasicCTUP* and *OptCTUP*, are proposed. These algorithms use a grid index to maintain the safety bounds of cells, and to update the top-k results by scanning the cells in the

descending order of the safety bounds. Algorithms can be terminated when the lower-bound score of the k-th candidate the higher than the upper bound score of all the other cells not scanned yet. The two algorithms are I/O efficient by maintaining the estimated bounds of cells in main memory instead of loading the contents of cells from the disk.

The CTUO query discussed in this chapter is more general, since objects to be protected can be either static or movable. Moreover, the safety weight of both protectors and protectees can be changed over the time. The two algorithms proposed in [62] cannot handle these generic situations, while algorithms proposed in this chapter work for both CTUP and CTUO queries.

Top-k Queries: Given a d-dimensional data set and a scoring function, a top-k query [30] returns the k records with the highest scores. The score of each record is related to the attributes of the record itself, so it describes the significance of characteristics of objects and only one relation is involved. Instead the CTUO query focuses on the relationship between two relations.

Many algorithms, like the Threshold Algorithm and the Non-Random-Access Algorithm [19] using bounds information, Onion [11] using convex hull, and PREFER [28] using materialized functions on partial scores, have been proposed. The same prune strategy used in the TA and NRA algorithms can be applied directly to the CTUO query, but it is quite inefficient, which is showed in Section 4.7.1

Continuous answering a top-k query can be processed by checking each update with the existing top-k objects and updating the top-k list accordingly. However this does not work when the main memory is limited and not all scored records can be maintained in main memory. [42] proposed an I/O efficient solution on monitoring top-k query results over a sliding window. Data are indexed using an axis-parallel grid and cells can be pruned during the top-k searching if the upper bound score of a cell is less

than the score of the k-th candidate.

Ranking Queries: There are also other variants of top-k queries related to the CTUO query, like rank-join [29] and rank-aggregate [38][59]. Both of them deal with scoring functions on multiple records instead of one record in the traditional top-k query. In these cases each record provides a *partial score* since it only contributes part of the final score. Rank-join returns the top-k join results for two given relations and a join condition. Rank-aggregate returns the top-k groups with highest aggregate values for a given relation and a grouping function. The CTUO algorithm proposed is novel since it combines both the join and aggregate into the same query, so existing algorithms working for rank-join and rank-aggregate cannot be applied directly to the CTUO query.

Continuous Location-based Queries: Many continuous location-based queries have been proposed, such as continuous k-means [63] and continuous k-NN [32]. Given a set of moving objects, a continuous k-means query monitors the k groups of objects by minimizing the total cost. The continuous nearest neighbor query returns the nearest neighbors of given objects when the locations of objects are changed over time.

The CTUO query proposed in this chapter cannot be solved by the algorithms proposed for existing continuous location-based queries. Queries mentioned above have a common property that the region of interest for each object is limited by the closeness on location. For example, in k-means the center assigned to an object should be closest to that object compared with other candidate centers, and in k-NN the relationship between the query point and the answer points should be "nearest". However in CTUO, since the protection region of a protector can be arbitrary, the processing region for each protectee is the whole data set instead of only the regions nearby. As a result, to maintain the answer of a CTUO query we need to process the whole space for every update.

4.3 Problem Definition

In this section we formally define the problem of continuously monitoring top-k unsafe moving objects (CTUO), and discuss its challenges.

4.3.1 Monitoring Top-k Unsafe Moving Objects

In monitoring top-k unsafe moving objects, two different categories of objects are involved: one is for objects to be protected (protectees), the other is for objects providing protection (protectors). To simplify the description, we use R_{PR} where "PR" is for "protection requirement" to represent the protectees, and R_{PS} where "PS" as "protection supply" for protectors. Data from the two classes have the schema as follows:

- If $r \in R_{PR}$, then r contains the following attributes: ID, timestamp, x location, y location, and safety requirement. Here *location* represents the location at the given timestamp, and *safety requirement* is a numerical value that specifies the requirement on protection.

- If $r \in R_{PS}$, then r contains the following attributes: ID, timestamp, x location, y location, safety supply, and coverage. The *safety supply* is the protection the protector can provide to protectees within its *coverage*, defined by the radius of the region it can protect around it.

So given $r_0 \in R_{PR}$ and $r_1 \in R_{PS}$, if the distance between r_0 and r_1 is not greater than r_1.coverage, then we say that r_1 is protecting r_0. The Euclidean distance is used throughout the chapter.

We use *safety weight*, or "sw" for short, to notate either the safety requirement for a protectee or safety supply for a protector. We use *unsafety weight* to represent the actual protection obtained by a protectee. Given an object r_0 from R_{PR}, its unsafety

weight is defined as the difference between its safety requirement and all protection provided by protectors covering it. Formally,

$$\text{unsafety}_{r_0} = r_0.\text{sw} - \sum_{r \text{ protects } r_0} r.\text{sw}$$

where the protection relation between $r \in R_{PS}$ and $r_0 \in R_{PR}$ is established as

$$\{r \text{ protects } r_0\} : Dist(r, r_0) \leq r.\text{coverage}$$

Note that in CTUO each protectee covered by a protector will get the protection equal to the maximum safety provided by the protector. This means that more protectees within the coverage region of a protector will not decrease the protection obtained by each protectee. We make this assumption since CTUO is aimed at preventing potential attacks instead of managing on-going multiple attacks. Therefore, we are interested in the case of a single attack in our framework. To deal with multiple concurrent attacks, the safety provided by a protector can be evenly distributed over protectees covered, and our proposed solutions are still feasible. However more complex cases, for example the case where the protection is distributed based on the distance between the protectee and the protector, are not covered in the chapter.

Thus our goal is to query the k most unsafe objects, which is equivalent to find the k objects with the highest unsafety weights. Now we define the top-k query on the most unsafe objects as follows, and an example on top-1 unsafe object query is showed in Figure 4.1.

Definition 3 *Top-k Unsafe Objects: Given two datasets, R_{PR} for protectees and R_{PS} for protectors, a* top-k query on most unsafe objects *returns the k objects having the*

highest unsafety weights.

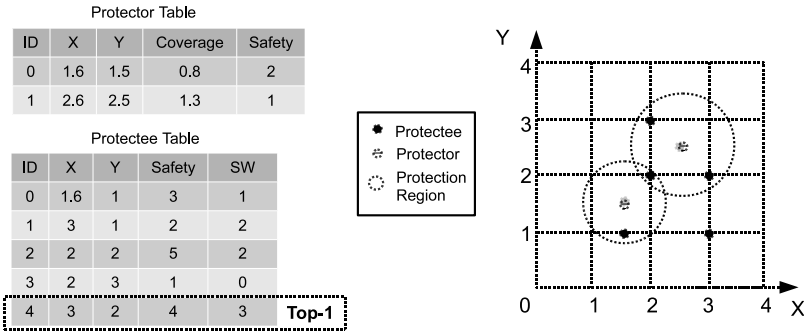


Figure 4.1: An example of the CTUO query.

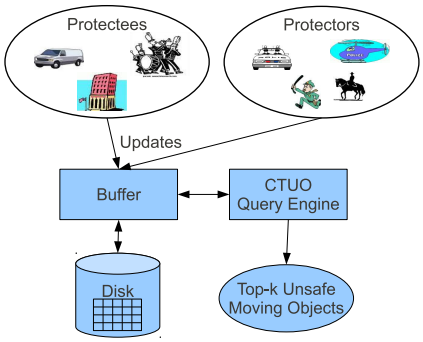


Figure 4.2: The query model for CTUO.

4.3.2 Continuous Processing Model

We extend Definition 3 into the continuous scenario, allowing dynamic updates to both the protectees and protectors data sets. A change has a timestamp, and all changes happen in increasing timestamp order. A change can be one of the following three cases.

- Create: A new object is created. This is triggered by an update coming with an id not in the two data sets. An object can be created only if it has never been seen in data sets or it has been deleted at some previous timestamp.

- Update: Attributes of an existing object are updated. Such an update may change the location, the safety weight, or the coverage.

- Delete: An existing object is deleted. Any further update with the same object id will be considered as a creation of a new instance of the object.

We say that an early version of an object is *invalidated* by either an update or a delete. Answers to top-k queries should be objects valid at the querying time, so the answer to the top-k query should be updated when the data set is updated.

Figure 4.2 shows the processing model for continuous monitoring top-k unsafe moving objects. Updates are implemented as a data stream fed into the system and stored on disk. The CTUO query engine loads data from disk for processing and returns the top-k results. In our framework we are interested in the disk I/O, since accessing data through disk I/O is the performance bottleneck in this model. So our goal is to reduce the disk I/O as much as possible.

4.3.3 Challenges of CTUO

There are three stages required to answer a CTUO query: *join, aggregate and sort*. In the join stage, for each protectee from R_{PR}, all protectors protecting it are retrieved to form a group. The join condition is defined by whether a protector can protect the given protectee within its protection coverage. In the aggregate stage, safety weights from protectors in the same group are aggregated and scores for groups are calculated. Finally in the sort stage, all protectees are sorted on their unsafety weights and the k highest ones are returned.

In order to get the unsafety weights for each protectee, each tuple in R_{PR} should be accessed at least once in order to identify the corresponding group. More

accesses to R_{PS} are necessary to maintain the scores of groups, because each tuple from R_{PS} may belong to several groups. This leads to a Cartesian product over the two relations for scores of all groups. Due to this reason we also refer the protectee relation to *target relation* and the protector relation as *support relation* for the analysis below.

There are more challenges when the CTUO query is processed in the continuous scenario. In order to maintain the correctness of the top-k results, every time when an update comes, intensive computation is required to update the groups influenced. In particular,

- The update is on R_{PR}: If such an update creates a new group or updates an existing group, maintaining the precise score of this group requires accesses to all tuples from R_{PS} to check the protection relationship. If any top-k group is influenced by the update, non-top-k groups may be processed to replace groups influenced, so an additional sort is necessary.

- The update is on R_{PS}: Groups influenced by this update should be updated, and the top-k list should be updated as well.

These challenges are amplified when the data sets to be processed are stored on disk. When disk accesses are necessary, a sort on all groups is computationally prohibited, while scans over data sets for join results cost even more. A naive approach requires a complete join every time an update arrives. The cost of the sort stage can be reduced by maintaining a top-k candidate list and sorting on only k objects at each update. However the join stage still costs large number of I/Os and makes the algorithm quite inefficient. To devise efficient algorithms, the number of joins and sorts should be controlled.

4.4 BoundPrune

One obvious approach to solve the CTUO query is to directly adapt the TA algorithm [19]. We call this approach BoundPrune and it is described below. Nevertheless, this approach has various disadvantages (as described in Section 4.5.1).

If we consider the safety weights of relations R_{PR} and R_{PS} in CTUO as the two scoring attributes, the TA algorithm works for CTUO too. Since the unsafety weight is positive related to the protector's safety weight while negative related to the protectee's safety weight, R_{PR} is sorted on the descending order of the safety requirement, while R_{PS} is sorted on the ascending order of the safety supply. Then the algorithm loads tuples from the two relations in the same way as the TA algorithm. We use $\bar{r}_{PR}$ and $\bar{r}_{PS}$ to representing the latest loaded tuple in R_{PR} and R_{PS}.

We still use the same idea to define the lower-bound and upper-bound in CTUO. Note that since the safety of a protectee is defined as the difference between its requirement and the protection supplies it can get, the lower-bound score of a protectee corresponds to the upper-bound safety weight provided by the protectors. So the upper-bound unsafety weight of a protectee can be calculated by the loaded protectors protecting the given protectee. Formally, for a given protectee r_0, for all $r_1 \in R_{PS}$ that $Dist(r_0, r_1) \leq r_1.coverage$,

$$ub_{r_0} = r_0.sw - \sum_{r_1 \text{ is loaded}} r_1.sw$$

In order to define the lower-bound of a score, we need to estimate upper bound of the protection from unknown protectors. However in CTUO, there may be many tuples from R_{RS} matching a single tuple from R_{PR} so the number of protectors from R_{PS} protecting a given protectee is unknown when R_{PS} is not fully processed. Without any

pre-knowledge about the group and the unloaded tuples in R_{PS}, we have to assume that all unloaded tuples in R_{PS} may protect the given protectee. If the depth of the latest loaded tuple in R_{PS} is d_{PS}, thus the lower-bound score of a given protectee r_0 can be formally defined as

$$lb_{r_0} = ub_{r_0} - (|R_{PS}| - d_{PS}) * \bar{r}_{PS}.sw$$

Algorithm 2 BoundPrune

Require: Two data sets R_{PR} and R_{PS}, k, F_G and F_S
Ensure: Return k groups with the highest scores

```
1: Sort the two data sets based on their partial score.
2: Initialize the top-k candidate list to be empty.
3: for each tuple r_0 ∈ R_PR do
4:    Initialize the lower bound and the upper bound of the group identified by r_0 as 0.
5: end for
6: for each tuple r_1j ∈ R_PS where j is the index of the tuple in the sorted order. do
7:    for each group g_i identified by r_0i ∈ R_PR do
8:       if F_G(r_0i, r_1) is TRUE then
9:          Update the lower-bound of g_i with the partial score of r_1.
10:      end if
11:      Update the upper-bound of g_i to be (g_i.lb + F_S(r_1j^{|R_PS|-j})).
12:      if G.score is larger than the score of the k-th candidate (0 if the candidate list
         has less than k groups) then
13:         Insert G into the top-k candidate list in order. Remove the groups with lowest
            scores to maintain only k candidates in the list.
14:      end if
15:   end for
16:   Remove groups outside of the candidate list has upper-bound larger than the
      lower-bound of the k-th group in the candidate list
17:   if no groups left outside of the candidate list then
18:      Return the candidate list as the final result.
19:   end if
20: end for
21: Return the top-k candidate list as the final result.
```

Algorithm 2 utilizes the bound pruning technique above. We use F_G to represent the join condition and F_S for the score function. Note that at Line 16 a protectee can be pruned if its upper bound score is no more than the lower bound score of the k-th candidate, which enables early termination if no other protectees are left for processing.

This is backed by the following lemma.

Lemma 4 *In the BoundPrune algorithm, for every protectee, its lower bound score will never be decreased, while its upper bound score will never be increased.*

Proof. Consider two timestamps t_1 and t_2 during the running time of the BoundPrune algorithm satisfying $t_1 < t_2$. The non-increasing upper-bound is easy to prove, since as the algorithm is processing, protectors already joined with the protectee will not be removed from the group, while there will at least 0 protector being added into the group. That is, for a given protectee r_0, for all $r_1 \in R_{PS}$ that $Dist(r_0, r_1) \leq r_1.coverage$,

$$\sum_{r_1 \text{ is loaded at } t_1} r_1.sw \leq \sum_{r_1 \text{ is loaded at } t_2} r_1.sw$$

which means $ub_{r_0}(t_1) \geq ub_{r_0}(t_2)$. The non-decreasing lower bound is based on the facts that as the algorithm is processing, 1) if the current processing protector does not cover the given protectee, then $lb_{r_0}(t_1) \leq lb_{r_0}(t_2)$ since its upper bound is not changed but the number of unloaded tuples in R_{PS}, which is $(|R_{PS} - d_P S|)$, will decrease; 2) if the current processing protector covers the given protectee, the new lower bound score will remain the same as follows.

$$\begin{aligned} lb_{r_0}(t_2) &= ub_{r_0}(t_1) - \bar{r}_{PS}.sw - (|R_{PS}| - d_{PS} - 1) * \bar{r}_{PS}.sw \\ &= ub_{r_0}(t_1) - (|R_{PS}| - d_{PS}) * \bar{r}_{PS}.sw \\ &= lb_{r_0}(t1) \end{aligned}$$

■ Based on the non-decreasing lower bound, we can deduce that the lower bound score of the k-th candidate is also non-decreasing. Since for a protectee outside of the candidate list, its upper bound score will never be increased, it can be pruned safely if

its upper-bound score is no more than the lower-bound score of the k-th candidate.

Unfortunately the BoundPrune algorithm is not very efficient. Our experiments in Section 4.7 show that when considering the loading depth of each relation, its I/O performance is almost the same as the naive algorithm. We will discuss this problem and our solutions to it in the next section.

4.5 GridPrune Algorithm

We propose an I/O efficient algorithm, called GridPrune, that utilizes the bound prune strategy mentioned above, and provides much tighter bounds compared with the BoundPrune algorithm. We firstly discuss the drawbacks of the BoundPrune algorithm, and then fix them in our proposed algorithm.

4.5.1 Drawbacks of BoundPrune

The reason for the poor performance of the BoundPrune algorithm is on the loose lower bound scores estimated. Recall that we define the lower bound score of a protectee using all the unloaded protectors and the maximum known safety weight from R_{PS}, which provides a safe but inefficient lower bound. In real applications, only a small part of the unloaded protectors from R_{PS} have protection relationship with a given protectee, so the size of the group is over-estimated. Moreover, the safety weights for unloaded protectors are over-estimated by the maximum safety weight seen so far. So during most of the running time the lower-bound score of the k-th candidate is too low to prune any groups outside of the candidate list.

Another problem for the BoundPrune algorithm is that all objects from R_{PR} must be accessed at least once. In the CTUO queries, we have the assumption that

the number of protectees is much greater than the protectors, so possible early prune can only happen on R_{PS}. Without being pruned before the termination, the cost to maintain the bounds information of the groups is huge.

BoundPrune is not I/O efficient either. Sorted accesses require more I/Os when the data sets are stored on disk. The algorithm requires data being sorted on partial scores, so location-based index structure cannot be used, and location information cannot be utilized. In particular, there are two kinds of reference locality when processing a CTUO query.

- Coverage Locality: Given a protector, within its coverage there may be more than one protectee that can be protected by it. I/O will be saved if these protectees can be loaded together from the disk storage into the main memory since they may be processed together if the protector is loaded.

- Update Locality: Given an update to a protector, its new location will be not very far away from its previous location given its movement is realistic over a map, thus there may be protectees covered by this protector in both timestamps. Again, I/O will be saved by keeping these protectees in main memory during the update.

4.5.2 GridPrune

The common property shared by the two locality optimization in CTUO is the *location closeness*. As a location based continuous query, CTUO has the property that the protection relationship between a protectee and a protector establishes based on their closeness, which is defined by the coverage attribute of a protector. Based on this property, protectees having similar locations (or close to each other) may also have similar bounds information, since the region containing these protectees may be covered

by the same group of protectors. So if the bounds information about this region is known, we can use them to estimate the bounds information of protectees inside.

The usage of region bounds can improve the BoundPrune algorithm with better pruning power. By grouping protectees on their locations, tighter bounds for their scores can be obtained compared with the BoundPrune algorithm. This is much tighter than considering all protectors not loaded in the whole data domain in the BoundPrune algorithm. Furthermore, a region can be pruned if its upper-bound score is lower than the k-th candidate, without loading and checking all protectees inside.

The BoundPrune algorithm can be further improved by accessing sorted cells instead of protectees. Since the bounds information of a cell can represent the bounds of protectees inside, cells with higher upper bounds of unsafety weight contain top-k unsafe protectees more probably.

Based on the observations above, we use the same grid to index both protectees and protectors. Protectees within the same cell have the similar location information and their bounds information can be estimated by the bounds of the cell. There are two advantages using such an index structure. First, since protectees with similar locations within a cell can be loaded together into the memory, I/O can be reduced by precaching. Second, the bounds information of a cell can be calculated without reading its content from the disk, so protectees in some cell can be pruned based on the cell bounds, which reduces the I/O by not loading the content of the cell.

In our implementation, for each cell in the grid, the following additional information is maintained in main memory:

- Protection Supply Lower-Bound: The minimum possible protection on this cell provided by protectors around it. This is measured as the sum of safety supply

Algorithm 3 GridPrune

Require: A grid index containing the valid objects from the two data sets R_{PR} and R_{PS}.

Ensure: Return k objects in R_{PR} with the highest safety weights.

Initialize three bounds values for each cell c_i: protection supply lower-bound slb_i, and protection requirement upper-bound rub_i.
Initialize influence cell list ICL_i for each cell c_i in the grid.
for each cell c_i in the grid **do**
  for each object o in c_i **do**
    if $o \in R_{PR}$ **then**
      Update rub_i if the safety requirement of o is higher than rub_i.
    else
      for each cell c_j in the grid **do**
        Update $c_j.slb$ if c_j is fully covered by o.
      end for
    end if
  end for
end for
for each cell c_i in the grid in the descending order of the value $(rub_i - slb_i)$. **do**
  if the unsafety weight of the k-th candidate is higher than the value of $(rub_i - slb_i)$ **then**
    Terminate the algorithm and return the candidate list as the top-k result.
  end if
  Load the cells in ICL_i.
  for each object $o \in R_{PR}$ in c_i **do**
    Calculate its precise unsafety weight by loading objects from the influencing cells
  end for
  Insert o into the candidate list in the descending order of the unsafety weight, if its unsafety weight is higher than the unsafety weight of the k-th candidate (0 if there are less than k candidates).
end for
Return the top-k candidate list as the final result.

from protectors whose coverage regions contain the whole cell.

- Protection Requirement Upper-Bound: The maximum protection requirement from protectees inside of this cell. This is measured as the sum of safety supply from protectors whose coverage regions at least intersect the cell.

- Influencing Cell List: A list of references pointing to cells containing protectors whose coverage regions intersect or cover this cell. This enables efficient calculation of unsafety weights of protectees in this cell without scanning all cells in the grid.

For a given cell, its protection supply lower-bound and influencing cell list can be calculated without inspecting its contents. Figure 4.3 shows the spatial relationships between a protection region and a cell. A protection region covers a cell if and only if all the four corners of the cell are covered by the protection region (Figure 4.3 A). A protection region intersects a cell if (i) at least one of the four corners of the cell is covered by the protection region (Figure 4.3 B), or (ii) none of the four corners is covered and at least one edge of the cell intersects with the protection region (Figure 4.3 C).

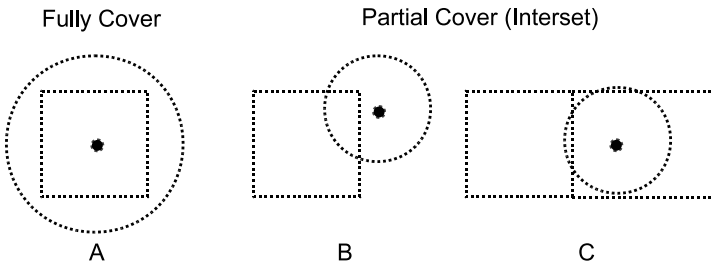


Figure 4.3: Spatial relationship between a coverage region and a cell: (A) Fully Cover; (B, C) Intersect.

Algorithm 3 shows the GridPrune algorithm. The whole algorithm consists of two stages. The first stage (Line 1 - 14) initializes the bounds information and the influence cell list for each cell, during which each cell is only scanned once. Then

the second stage (Line 15 - 24) accesses cells in the descending order of their unsafety weight upper-bound. When accessing a cell, protectees contained are loaded and their unsafety weights are calculated by loading the influence cells of this cell. Protectees whose unsafety weights are larger than the score of the k-th candidate are inserted into the candidate list. The algorithm can be terminated if the next visiting cell has its score upper-bound no more than the score of the k-th candidate, or all cells have been inspected. Finally the candidate list is returned as the top-k result.

Although the GridPrune algorithm mainly works for a static query, it can also be applied to the continuous environment where *batch updates* are allowed, while reserving its efficiency. That is, before another top-k query is issued, there are many updates streamed into the system. In such case, the cost of re-calculating the bounds information and also updating the top-k candidate list can be amortized to multiple updates, which will decrease the total I/O cost. In Section 4.7 the comparison between GridPrune and another proposed algorithm, GridPrune-Pro, will discuss this point in more detail.

Compared with the CTUP algorithm proposed in [62], the GridPrune algorithm maintains the exact bounds information of each cell instead of the estimated ones. This increases the memory requirement on the influence cell lists, however it avoids the "flashing cell problem" in CTUP [62] where the bounds information of a cell may be updated (increased or decreased) unnecessarily. Although the OptCTUP algorithm proposed in [62] solves this problem by using hash functions to record the protectors counted already, theoretically the cost of maintaining a hash over influencing objects is the same as the cost for keeping these influence cell lists, or even worse (multiple protectors in the same cell can be maintained by a single cell in GridPrune instead of one hash key for each).

4.6 GridPrune-Pro Algorithm

The GridPrune algorithm proposed in the previous section does not work efficiently for the continuous situation. To handle an update, recalculating the bounds information of each cell is still I/O expensive. Furthermore, the update may not change the top-k result from the previous timestamp so the second stage of the GridPrune algorithm is not always necessary. According to the update locality, the unsafety weight of a protectee may not change if the protector covers it in both two timestamps, and the safety provided by the protector is not changed either.

We now propose a progressive algorithm of GridPrune, called *GridPrune-Pro*, to handle the dynamic updates on both protectees and protectors. The main algorithm is divided into two parts, one is to update the bounds information and the other is to update the top-k list. The second part can be invoked any time a top-k result is needed, and it is calculated based on the previous top-k results and the updated bounds information, so it is much more efficient than re-running the GridPrune algorithm.

Once an update is issued, depending on whether it is for a protector or a protectee, different strategies are used to maintain the updated bounds information for influenced cells as follows (implemented in Algorithm 4).

- Update a protector: The protection supply lower-bounds of cells influenced by the update are adjusted based on the spatial relationship described in previous section. Cells influenced by the previous version of the protector but not the new one should remove the corresponding cell (the cell containing the previous version of the protector) from their influence cell lists.

- Update a protectee: If the protectee stays in the same cell after update, disk access is necessary only if the update changes the protection requirement upper bound

of the cell. If the protectee is not in the same cell after update, the protection requirement upper bounds of the two cells need to be updated if they are changed due to the update. The protection requirement upper bound of a cell may be changed if the protectee with the highest safety requirement is updated to a lower safety requirement, or a new protectee with a higher safety requirement than the safety requirement upper bound of the cell is inserted.

Algorithm 4 GridPrunePro: Bounds Update

Require: A grid index containing the valid objects from the two data sets R_{PR} and R_{PS}, an update u

Ensure: Update the grid index, and also bounds information maintained for top-k processing.

1: Insert u in the grid index and invalidate the previous version of object o where $o.id = u.id$.
2: Load the previous version of o, and let it be o_p
3: **if** u is an update on an object $o \in R_{PS}$ **then**
4: **for** each cell c in the grid **do**
5: **if** $o_p \in c.ICL$ **then**
6: Adjust $c.slb$ and $c.sub$ by removing o_p from the corresponding list.
7: **end if**
8: **end for**
9: Calculate the cell c' containing u
10: Update $c.slb$ if c is fully covered by o.
11: Update $c.sub$ if c is at least partial covered by o. Add c' into the influence cell list ICL of c.
12: Adjust scores of the top-k candidates using o_p and u.
13: **else**
14: Adjust rub of the cell containing o_p.
15: Adjust rub of the cell containing u.
16: Update top-k candidate list, if u is contained in the candidate list.
17: **end if**

The top-k list is then updated on demand when Algorithm 5 is invoked. Intuitively, the top-k candidate list will be updated only if there are cells whose score upper-bounds are higher than the score of the k-th candidate. In details, this may happen if (i) the score upper-bound of a cell is increased after update, or (ii) the update has influence to the top-k candidate list so that the score of the k-th candidate is decreased,

or some top-k candidate is removed from the top-k list (which will decrease the score of the k-th candidate to 0).

Notice that Algorithm 5 does the same job as the second stage of the GridPrune algorithm. However in GridPrune-Pro, previous top-k results are reused so cells with their score upper-bounds lower than the score of the k-th candidate can be pruned earlier than the GridPrune algorithm.

Algorithm 5 GridPrunePro: Top-k Update

Require: A grid index containing the valid objects from the two data sets R_{PR} and R_{PS}, an update u

Ensure: Update the top-k list from the current bounds information

1: **for** each cell c_i in the grid in the descending order of the value $(rub_i - slb_i)$. **do**
2: **if** the unsafety weight of the k-th candidate is higher than the value of $(rub_i - slb_i)$ **then**
3: Terminate the algorithm and return the candidate list as the top-k result.
4: **end if**
5: Load the cells in ICL_i.
6: **for** each object $o \in R_{PR}$ in c_i **do**
7: Calculate its precise unsafety weight by loading objects from the influencing cells
8: **end for**
9: Insert o into the candidate list in the descending order of the unsafety weight, if its unsafety weight is higher than the unsafety weight of the k-th candidate (0 if there are less than k candidates).
10: **end for**
11: Return the top-k candidate list as the final result.

4.7 Performance Analysis

We implement the proposed algorithms in Java running on Mac OS X 10.6.3 (Snow Leopard), on a MacBook Pro with a Intel 2.2 GHz Core 2 Duo processor and 4 GB memory. For the disk-based grid index structure, we use the NEUStore package [61] to simulate the disk and buffer activities during the query processing.

We use two synthetic data sets in our tests, one is only for static query, and

the other is for both static and continuous cases. For the first data set, we generate objects with randomly selected types (protectee or protector), location, safety weight and coverage (only for protector). For the second data set, we use the Network-based Generator of Moving Objects [9] to generate simulating objects moving along the road network of Oldenburg.

4.7.1 Naive v.s. BoundPrune

Firstly we measure the prune power from the BoundPrune algorithm compared with the naive implementation (join-and-sort approach). Two important factors are measured in an in-memory simulation, the depth of data loaded for each relation, and the size of data loaded before the algorithm is terminated. Figure 4.4 shows the prune power of the BoundPrune algorithm, where the maximum upper-bound (blue curve) and the lower-bound of the k-th candidate (red curve) are plotted on the size of the data loaded. Two different data loading strategies are tested. When the target relation (protectee set) is loaded at first, the bounds of groups are not changed until the support relation starts loading. When the two data sets are loaded in a round-robin fashion, the difference between bounds is shrinking as the size of the data loaded is increasing, however the speed is slower than the "group-first" strategy after loading the target relation. Both cases take a long time before the early termination is satisfied.

Figure 4.5 shows the percentage of data loaded and join operations in the BoundPrune algorithm compared with the naive algorithm where all data should be loaded. Different size of the two relations are used ($|R_0| >> |R_1|, |R_0| = |R_1|$ and $|R_0| << |R_1|$). In all three cases the save on the data loaded is no more than 2%, since the algorithm cannot be terminated earlier with loose bounds. So the I/O cost

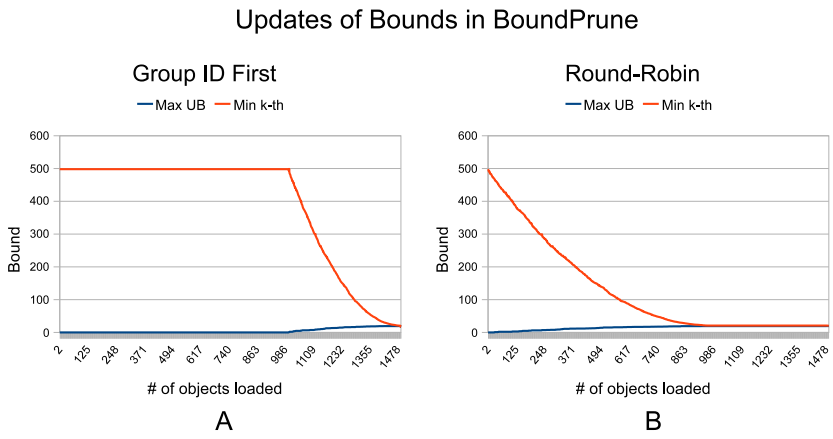


Figure 4.4: Changes on bounds in the BoundPrune algorithm: The score of the k-th candidate (red curve), and the maximum upper-bound score of groups outside of the candidate (blue curve).

of the BoundPrune algorithm is almost the same as the one using the naive algorithm. Compared with the data loaded, join operations saved in the BoundPrune algorithm are at most 15%, and the most save happens when the support relation is smaller than the target relation, because it leads to a tighter bounds when we consider the size of the unloaded data in the bounds information.

4.7.2 GridPrune v.s Naive

Since the BoundPrune algorithm is not suitable for a grid index, and its I/O performance is almost the same as the naive algorithm, we run experiments to compare the performance between the naive algorithm and the GridPrune algorithm. In the naive algorithm, all cells are scanned and a complete Cartesian product is processed every time an update comes. For an update in the GridPrune algorithm, the top-k candidate list will be cleared and generated from the latest bound information again.

Figure 4.6 shows the experimental results. Two measurements are used to represent the I/O performance. *Disk Reads* shows the count on the disk read requests

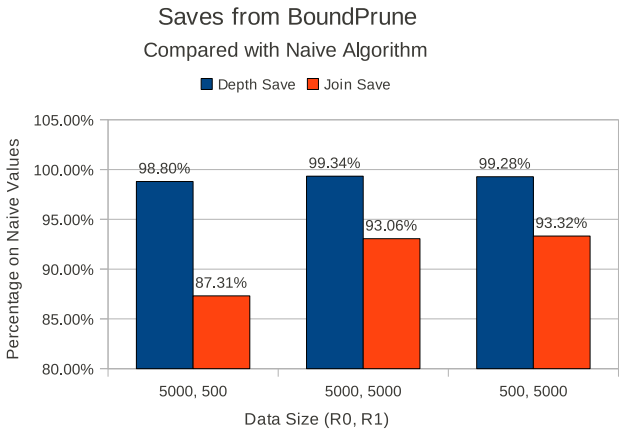


Figure 4.5: Performance gains using BoundPrune compared with the Naive algorithm on load depths and join counts.

issued by the program, while *Buffer Reads* is an counter for the buffer read requests. Buffer reads should be always larger than disk reads since not every buffer read correspond to a disk read. The write I/O is omitted since only the reads are important for queries. From the figures clearly the GridPrune algorithm outperforms on both the two measurements. We can also see that with the increase of updates over the time, the disk read I/O and also the buffer read requests in naive algorithm increases very fast.

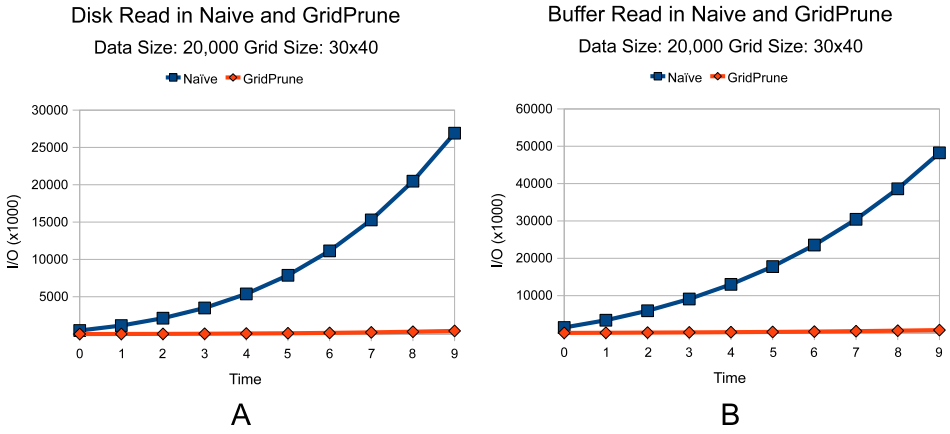


Figure 4.6: Disk reads (A) and buffer reads (B) in Naive and GridPrune algorithms.

The number of the grid cells also has influence on the performance of Grid-

Prune. From Figure 4.9 we can see that when the number of cells is small (e.g. each cell is large, 30×40 in Figure 4.9), the cost can be reduced by increasing the number of cells (60×80 and 120×160 in Figure 4.9) in order to have better prune capability. However when the size of a cell is too small (300×400 in Figure 4.9), the cost increases again for more I/O cost to maintain the bounds during updates. The GridPrunePro algorithm also has the same property.

4.7.3 GridPrune-Pro v.s. GridPrune

Figure 4.7 shows the I/O performance comparison between GridPrune and GridPrune-Pro when the grid size is 300×400. GridPrune-Pro shows better performance on both disk reads and buffer reads, since it progressively maintains the candidate list when updates are streamed in. GridPrune-Pro is much better in a long time running as showed in Figure 4.8, where the disk reads are increased faster in GridPrune than GridPrune-Pro.

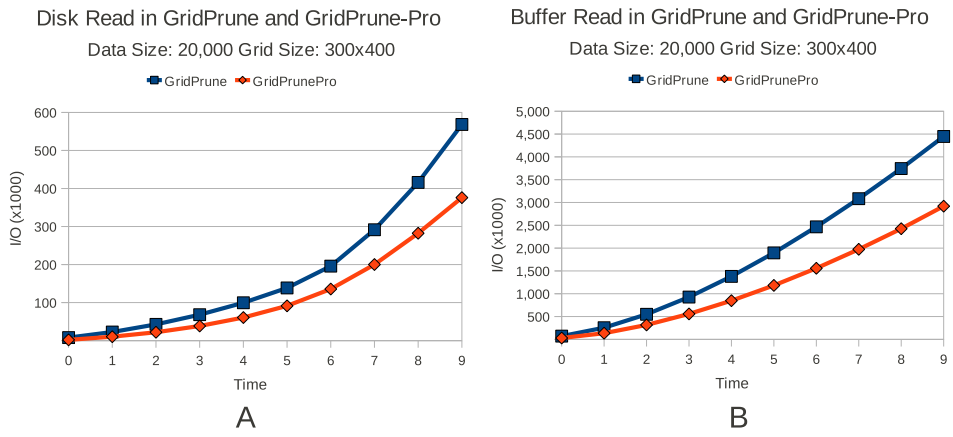


Figure 4.7: Disk reads (A) and buffer reads (B) in GridPrune and GridPrune-Pro algorithms.

The I/O performance of the GridPrune-Pro algorithm is not always better

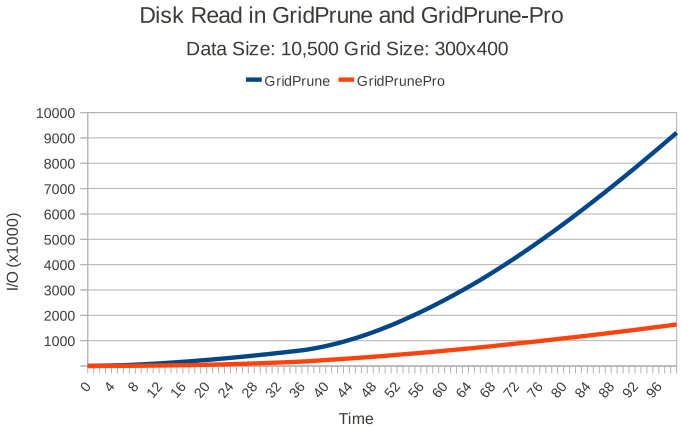


Figure 4.8: Disk reads in GridPrune and GridPrune-Pro algorithms in long-time running example (large number of cells).

than the GridPrune algorithm. Recall that GridPrune-Pro is working by updating only necessary cells influenced by updates. When the number of cells is small, it is possible that most of the cells are influenced by updates so its I/O cost should be more, so in order to maintain the correct bounds information for each cell, the I/O cost for each update is high. On the contrary GridPrune can be used to handle updates in bundle so the I/O cost for a total rerun can be amortized to multiple updates. Figure 4.9 illustrates this phenomena. Figure 4.10 shows that in a long-time running example, the disk read cost of the GridPrunePro algorithm increases faster, due to its overheads on maintaining the bounds information for every cell when the number of cells is small.

4.8 Summary

In this chapter we discussed the continuous top-k query on most unsafe moving objects or CTUO. The basic prune strategy using bounds information is discussed. Two I/O efficient algorithms, GridPrune and GridPrune-Pro, are proposed to utilize the prune strategy using grid index. Experimental results show the good I/O performance of

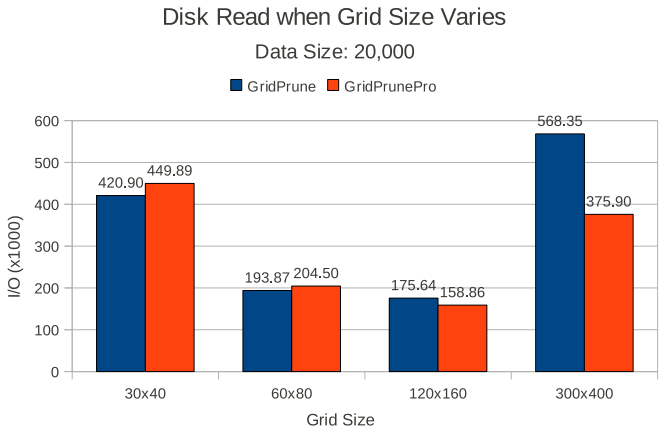


Figure 4.9: Disk reads in GridPrune and GridPrunePro when the grid size varies.

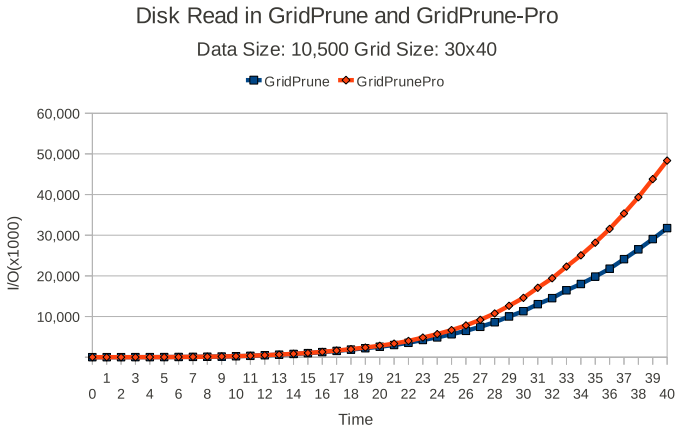


Figure 4.10: Disk reads in GridPrune and GridPrune-Pro algorithms in long time running example (small number of cells).

the proposed algorithms compared with the naive algorithm. Furthermore, GridPrune is efficient when the number of grid cells is small, or in dynamic case where batch updates are required. GridPrunePro shows good I/O performance in dynamic case when the number of grid cells is large.

There are several future topics inspired by the CTUO query. Firstly, the two proposed algorithms are based on the grid index to utilize the prune capability on bounds of cells. If the same idea can be applied to other spatial index structures like R-trees, the bound prune strategy can be easily applied to existing database systems no matter what kind of index structures they are using. Secondly, it is interesting to think about its generic query form, where the join condition and the aggregate function may be arbitrary but not only on the location closeness. This generic query is attractive because it tries to measure the top-k significant relationships between data sets, and there should be a big application area on this query.

5

Conclusions

In this thesis we revisited and discussed the aggregation operation in a big data environment. The important requirements for an aggregation implementation from the big data include: (1) the widely usage of the commodity cluster (like Hadoop) require that an aggregation algorithm implementation should be robust for very strict resource budget; (2) various kinds of data could be streamed into the system for aggregation, and it is important for the aggregation processing to be adaptive for different input data; (3) to utilize the multiple node resource for parallel aggregation, the physical plan should be properly picked in order to maximize the resource utilization; (4) aggregation processing for specific applications should consider the characteristics of the application and the data for efficient processing techniques.

We have addressed these issues separately in the chapters of this thesis. We first discussed the implementation details and also a generic but precise cost model for some popular local aggregation algorithms. Then this local aggregation cost model is further extended to predicate the global aggregation cost. Through this hardware independent cost models, we provide the opportunity for the query optimizer of a big

data system to pick the most efficient plans based on the cost model. We verified our models and algorithms through extensive experiments. Finally we have discussed a new aggregation problem over the spatial and temporal information enriched data set, and proposed the efficient algorithms utilizing indexing structures with the spatial and temporal information.

Although we have managed to cover as much as possible for the aggregation works, there are still lot of space to approach for the aggregation study. What we have described here would be able to provide a generic framework as a starting point for more advanced aggregation processing techniques. Also, we were focusing on only the shared-nothing environment, however it will be very interesting to study whether the assumptions for a shared-nothing environment still holds for other environment like the multi-core environment, cloud environment (heterogeneous network structures). we sincerely hope that the work discussed in this thesis will be helpful for the future research in the aggregation field.

Bibliography

[1] Asterixdb: http://asterixdb.isc.uci.edu.

[2] Smhasher. http://code.google.com/p/smhasher.

[3] Tpc-h benchmark. http://www.tpc.org/tpch.

[4] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *GPGPU*, pages 94–103, 2010.

[5] A. Behm, V. R Borkar, M. J Carey, R. Grover, N. Onose C. Li, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[6] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distrib. Parallel Databases*, 29(3):185–216, June 2011.

[7] D. Bitton, H. Boral, D. DeWitt, and K. Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst.*, 8(3):324–353, 1983.

[8] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[9] Thomas Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[10] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[11] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: indexing for linear optimization queries. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 391–402, New York, NY, USA, 2000. ACM.

[12] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[13] J. Cieslewicz and K. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[14] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD Conference*, pages 1115–1118, 2010.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[16] D. DeWitt. The wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook*. 1993.

[17] R. Epstein. Techniques for processing of aggregates in relational database systems. Technical report, Technical Report UCB/ERL, 1979.

[18] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[19] Ronald Fagin. Combining fuzzy information from multiple systems (extended abstract). In *PODS '96: Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 216–226, New York, NY, USA, 1996. ACM.

[20] D. Gardy and L. Némirovski. Urn models and yao's formula. In *ICDT*, pages 100–112, 1999.

[21] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[22] G. Graefe. The value of merge-join and hash-join in sql server. In *VLDB*, pages 250–253, 1999.

[23] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft sql server. In *VLDB*, pages 86–97, 1998.

[24] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 152–159. IEEE Computer Society, 1996.

[25] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

[26] Stavros Harizopoulos and Qiong Luo, editors. *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*. ACM, 2011.

[27] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In *VLDB*, pages 656–667, 2003.

[28] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 259–270, New York, NY, USA, 2001. ACM.

[29] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 754–765. VLDB Endowment, 2003.

[30] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):1–58, 2008.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[32] Glenn S. Iwerks, Hanan Samet, and Ken Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 512–523. VLDB Endowment, 2003.

[33] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[34] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[35] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[36] Bhaskar Krishnamachari, Deborah Estrin, and Stephen B. Wicker. The impact of data aggregation in wireless sensor networks. In *ICDCS Workshops*, pages 575–578, 2002.

[37] Per-Åke Larson. Data reduction by partial preaggregation. In *ICDE*, pages 706–715, 2002.

[38] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 61–72, New York, NY, USA, 2006. ACM.

[39] Xuemin Lin, Yidong Yuan, Qing Zhang, and Ying Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.

[40] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.

[41] R. Motwani and P. Raghavan. Randomized algorithms. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 141–161. CRC Press, 1997.

[42] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 635–646, New York, NY, USA, 2006. ACM.

[43] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[44] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178. ACM, 2009.

[45] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[46] Joshua Rosen, Neoklis Polyzotis, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Iterative mapreduce for large scale machine learning. *CoRR*, abs/1303.3517, 2013.

[47] R. Sedgewick. Quicksort with equal keys. *SIAM J. Comput.*, 6(2):240–268, 1977.

[48] R. Sedgewick and J. Bentley. Quicksort is optimal.

[49] L. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[50] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD Conference*, pages 104–114, 1995.

[51] Ambuj Shatdal and Jeffrey F. Naughton. Processing aggregates in parallel database systems. Technical report, University of Wisconsin-Madison, Computer Sciences Department, 1994.

[52] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

[53] R. Snodgrass and M. Winslett, editors. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*. ACM Press, 1994.

[54] Markus Weimer, Sriram Rao, and Martin Zinkevich. A convenient framework for efficient parallel multipass algorithms. In *In LCCC : NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, 2010.

[55] Jian Wen, Vassilis J. Tsotras, and Donghui Zhang. On continuously monitoring the top-k moving objects with relational group and score functions. *The SIGSPATIAL Special*, 1(3), November 2009.

[56] Tian Xia and Donghui Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 77, Washington, DC, USA, 2006. IEEE Computer Society.

[57] Tian Xia, Donghui Zhang, Zheng Fang, Cindy X. Chen, and Jie Wang. Online subspace skyline query processing using the compressed skycube. *ACM Trans. Database Syst.*, 37(2):15, 2012.

[58] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9, 2011.

[59] Man Lung Yiu, Nikos Mamoulis, and Vagelis Hristidis. Extracting k most important groups from data efficiently. *Data Knowl. Eng.*, 66(2):289–310, 2008.

[60] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.

[61] Donghui Zhang. Neustore: A simple java package for the construction of disk-based, paginated, and buffered indices. 2005.

[62] Donghui Zhang, Yang Du, and Ling Hu. On monitoring the top-k unsafe places. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 337–345, Washington, DC, USA, 2008. IEEE Computer Society.

[63] Zhenjie Zhang, Yin Yang, Anthony K. H. Tung, and Dimitris Papadias. Continuous k-means monitoring over moving objects. *TKDE*, 2008.

Appendix A

Cost Model Components

This appendix describes the details of the basic component models used in the cost model analysis. We use the symbols shown in Table A.1.

Symbol	Description
n	Number of raw records
m	Number of unique groups
A	A set of run files $\{A[1], ..., A[\|A\|]\}$
$\mathcal{D}(n, m)$	An input dataset of n records and m unique groups
H	Hash table slots count
K	Hash table capacity in number of unique groups
M	Memory capacity in frames
$\mathcal{U}$	Number of unique keys, so that the dataset $\mathcal{D}(n, m)$ can be generated through with-replacement draws from this key set.

Table A.1: Symbols For Input Parameters

A.1 Input Component

There are two important quantities we will use in the algorithms' cost models. (1) Due to a restricted memory budget, a dataset $\mathcal{D}(n, m)$ will be processed in 'chunks'. When considering a chunk of r records ($r \leq n$), an important quantity is the

number of unique keys that this chunk contains - denoted as $I_{key}(r, n, m)$ - assuming that the records are randomly picked from $\mathcal{D}(n, m)$. (2) Given a memory budget for k groups (records of the form (`key`, `aggregated value`)), another important quantity is the number of records - denoted as $I_{raw}(k, n, m)$ - that we should pick randomly from $\mathcal{D}(n, m)$ in order to fill up the memory with k unique keys ($k \leq m$). Assuming draws without replacement, both quantities can be computed through direct application of Yao's formula [20]. In particular:

$$I_{key}(r, n, m) = m * (1 - (1 - \frac{r}{n})^{\frac{n}{m}}) \tag{A.1}$$

$$I_{raw}(k, n, m) = n * (1 - (1 - \frac{k}{m})^{\frac{m}{n}}) \tag{A.2}$$

A.2 Sort Component

When sorting the dataset $\mathcal{D}(n, m)$, we assume a 3-way-partition-quicksort [47]. The required number of comparisons $C_{sort}(n, m)$ can be computed through a divide-and-conquer procedure by randomly choosing a split key and recursively sorting on the two sub-partitions:

$$\begin{aligned} C_{sort}(n, m) &= \frac{n}{m} * m - 1 \\ &+ \frac{1}{m} \sum_{i=1}^{m} (C_{sort}(\frac{n}{m} * (m - i), m - i) + \\ &\quad C_{sort}(\frac{n}{m} * (i - 1), i - 1)) \end{aligned} \tag{A.3}$$

Solving this recurrence we get the following formula:

$$C_{sort}(n,m) = 2\frac{n}{m}(m-1)ln(m-2) + (\frac{n}{m}-1)(2m-3) \quad \text{(A.4)}$$

A.3 Merge Component

Consider a collection A of sorted run files. Let $A[i]$ denote the size of the i-th file. Algorithm 6 computes the cost for merging the collection A using M input buffer frames and the loser-tree based merging method [35]. By setting the cost function $F(A')(A' \subseteq A)$ to be the CPU comparisons in merging ($F(A') = log_2(|A'|)$) or the flushing I/O in merging ($F(A') = \sum_{i=1}^{|A'|} A'[i]$), the same algorithm can be used for either CPU comparison cost or the I/O cost.

Algorithm 6 Algorithm for Merge Cost

Require: A: files to be merged; M: available memory in frames; F: cost function.
  while $|A| > 1$ **do**
    if $|A| \leq M$: all files can be merged in a single round. Add $F(A)$ to cost, and stop.
    if $M < |A| < 2M$: merge the first $(|A| - M + 1)$ files to produce a single run; remove the merged files from A and add the new run at the end of A ($|A|$ is thus reduced to M files). Add $F(\{A[1], ..., A[|A| - M + 1]\})$ to $C_{merge}(A, M)$
    if $|A| \geq 2M$: merge the first M files into a new run. Remove the merged files from A and add the new run at the end of A. Add $F(\{A[1], ..., A[M]\})$ to $C_{merge}(A, M)$
  end while

A.4 Hash Component

Consider a hash table with H slots in the slot table; Let K denote the maximum number of group records that can be stored in the list storage area. Note that duplicates are aggregated within group records, so filling up the list storage area would imply encountering K unique groups. The number of records drawn randomly from $\mathcal{D}(n,m)$ to fill up the list storage area (i.e., to get K unique keys) is thus $I_{raw}(K,n,m)$.

Let $C_{hash}(n,m,K,H)$ denote the number of comparisons needed to fill up the

list storage area. This accounts for both hash hits (denoted as c_{succ}; these are records that have been seen already and are thus aggregated) and hash misses (denoted by c_{unsucc}; these are records that have not been seen before). For the i-th insertion to be a hash hit, it must correspond to a key which has already been inserted in the hash table. Using Equation A.1, at the i-th insertion the number of unique keys already in the hash table is

$$k_i = I_{key}(i, n, m)$$

Note that the no-replacement assumption of Yao's formula implies that after each insertion, the distribution of the remaining keys in the input set changes; this distribution is thus difficult to re-estimate after each insertion. Instead, we will assume here that the dataset $\mathcal{D}(n, m)$ is generated by randomly drawing keys **with replacement** from a 'generator' set with $\mathcal{U}$ unique keys. Then each insertion can be considered as a random pick from the $\mathcal{U}$ unique keys with replacement, and the probability for a hash hit for the i-th insertion becomes:

$$Pr_{hashHit} = \frac{k_i}{\mathcal{U}}$$

We note that the average number of unique keys $\tilde{m}$ in n random draws is given by:

$$\tilde{m} = \mathcal{U} * (1 - (1 - \frac{1}{\mathcal{U}})^n) \tag{A.5}$$

We can then estimate $\mathcal{U}$ by substituting the expected value $\tilde{m}$ with m in the above equation.

To compute the number of comparisons during a hash hit we need the expected number of groups contained in a non-empty slot, assuming the probability of finding a

match at any group along the slot's linked list is the same. The number of non-empty slots in the hash table at the i-th insertion can be calculated using the urn model [41] as

$$H_u(i, H, n, m) = H * (1 - (1 - \frac{1}{H})^{k_i}) \tag{A.6}$$

Then the expected number of groups in a non-empty slot would be

$$L_{slot} = \frac{k_i}{H_u(i, H, n, m)}$$

The expected comparison cost for a hash hit becomes:

$$c_{succ}(i, n, m, H) = Pr_{hashHit} * \frac{L_{slot} + 1}{2} \tag{A.7}$$

A hash miss happens when a record is hashed either into a previously empty slot (this does not require a comparison) or into a non-empty slot but where no match is found (this case will incur comparisons until the end of the linked list is reached). The probability that it is inserted into a non-empty slot is:

$$Pr_{nonempty} = \frac{H_u(i, H, n, m)}{H}$$

and thus the hash miss comparison cost then becomes:

$$\begin{aligned} c_{unsucc}(i, n, m, H) &= (1 - Pr_{hashHit}) \\ &* \quad L_{slot} * Pr_{nonempty} \end{aligned} \tag{A.8}$$

Finally, the total comparison cost is given by:

$$C_{hash}(n,m,K,H) = \sum_{i=0}^{I_{raw}(K,n,m)} (c_{succ}(i,n,m,H) + c_{unsucc}(i,n,m,H)) \quad \text{(A.9)}$$

For some hybrid-hash algorithms a spilled partition may contain both aggregated groups and non-aggregated records. To insert such a "mixed" dataset into a hash table, the cost model should be adjusted. Let u denote the number of aggregated groups (which are thus unique) and n the number of 'raw' (not yet aggregated) records. To insert the u unique groups the comparisons arise only from hash misses:

$$c_{unique}(n,m,H,u) = \sum_{i=1}^{u} (\frac{H_u(i-1,H,n,m)}{H} * \frac{i-1}{H_u(i-1,H,n,m)}) \quad \text{(A.10)}$$

For calculating the number of comparisons from the insertion of the raw records after inserting the u unique groups, we first note that the probability for a hash hit is:

$$Pr'_{hashHit} = \frac{k_i + u}{\mathcal{U}}$$

The expected number of groups in a non-empty slot for the i-th insertion is given by:

$$L'_{slot} = \frac{k_i + u}{H_u(i,H,n,m)}$$

So the hash comparison cost if the i-th insertion is a hash hit is:

$$c_{succ}(i,n,m,H,u) = Pr'_{hashHit} * \frac{L'_{slot}+1}{2} \tag{A.11}$$

To calculate the hash miss cost, we note that the probability that the insertion is to a non-empty slot is adjusted as:

$$Pr'_{nonempty} = \frac{H_u(i+u,H,n,m)}{H}$$

Hence the comparison cost for the hash miss of the i-th insertion becomes:

$$c_{unsucc}(i,n,m,H,u) = L'_{slot} * Pr'_{nonempty} * (1 - Pr'_{hashHit}) \tag{A.12}$$

The overall cost for the 'mixed' input case, denoted by $C_{hash}(n,m,K,H,u)$ is thus:

$$\begin{aligned} C_{hash}(n,m,K,H,u) &= c_{unique}(n,m,H,u) \\ &+ \sum_{i=1}^{I_{raw}(K-u,n,m)} (c_{succ}(i,n,m,H,u) \\ &\quad + c_{unsucc}(i,n,m,H,u)) \end{aligned} \tag{A.13}$$