Taming Very Large Data in Analytical Workloads

Mohammed ElSeidy, Ph.D. Student
DATA, I&C, EPFL

***Abstract*—A wide class of data analytics operations can be modeled as classical database join operations. In the current era of data deluge, the explosion in data sizes has led to the introduction of parallelization as a prominent solution towards early query termination and attaining high throughput. Unfortunately, parallelization is not sufficient to guarantee systems with reasonable operational latency. Online algorithms can be used to reduce latency, at the cost of sacrificing accuracy. Early approximate results are useful indicators of the "big picture" of the underlying data and are smoothly refreshed to exact results. We discuss prior work done towards online aggregation and statistical estimates. We begin with ripple joins [6]; an online family of join algorithms that produce running estimates for the query's result defined within confidence intervals. Later on, we describe the DBO system [8] which is designed to overcome the scalability limitations of ripple joins. To further support arbitrary join conditions, we analyze an efficient parallel offline MapReduce algorithm [11] that supports any Θ-join condition. Finally, we conclude that taming these vast sizes of data requires an online parallel solution. We discuss the challenges faced in supporting general join conditions and parallelism in an online fashion.**

***Index Terms*—Parallel joins, Theta joins, Non-equi joins, Non-blocking joins, Online joins.**

Proposal submitted to committee: June 30th, 2012; Candidacy exam date: September 3rd, 2012; Candidacy exam committee: Prof. Willy Zwaenepoel, Prof. Christoph Koch, Prof. Anastasia Ailamaki.

This research plan has been approved:

Date: ____________________

Doctoral candidate: ____________________
(name and signature)

Thesis director: ____________________
(name and signature)

Thesis co-director: ____________________
(if applicable) (name and signature)

Doct. prog. director: ____________________
(R. Urbanke) (signature)

I. Introduction

DEEP data analytics play an essential role in marketplace competitiveness, as they represent a cornerstone for businesses' prosperity. Sophisticated analysis and data mining aid decision making in operations management, which in turn result in cost savings and direct revenues for companies. Joins are one of the fundamental operations in relational database but computationally expensive and resources-intensive. Joins play a great role in data analysis, representing the bulk of the operations of analysis processing. Furthermore, it was shown that joins are utilized to support vector and matrix operations, e.g., matrix addition and multiplication [3]. Provided that these operations represent the basic building blocks for statistical analysis, efficiently supporting joins (with arbitrary conditions) is essential for the feasibility of deep data analysis.

Data analytics target very large sizes of historical data such as click-streams, software logs, email and discussion forum archives. This data might be gathered from a plethora of operational databases or from large-scale experiments and sensors (e.g. Large Hadron Collider, National Virtual Observatory) where the data sizes are petabytes or even exabytes. In response to such data deluge, a prominent solution was to scale out. MapReduce [4], has emerged as one of the most popular paradigms for parallel computation. MapReduce has greatly impacted data management research, e.g., devising MapReduce join algorithms. Although these algorithms are parallel and efficient, they were initially rather restrictive in terms of the supported join conditions that can be queried with. In [11], *Okcan et al.* propose a mechanism to evaluate Θ-joins over MapReduce; a solution that can evaluate queries with arbitrary join conditions while still exploiting the parallelism, scalability and fault tolerance of MapReduce.

Batch processing systems, like MapReduce, are optimized for high throughput in contrast to low latency, thus the user has to wait until the very end of a query to yield exact results. Datasets, being too large, typically require hours or even days to run a query to the end. This is in stark contrast to the common requirement of low latency imposed by the industry, where interactivity and quick decision making are vital. A representative example may be an analyst's workflow. An analyst tries to gain preliminary insights at very early stages. The data is quickly examined for a "rough picture" by running various queries. Approximate or incomplete results at this stage usually suffice. Other examples include real-time applications such as updating Twitter's back-end systems to incorporate new tweets or to compute real-time analytics.

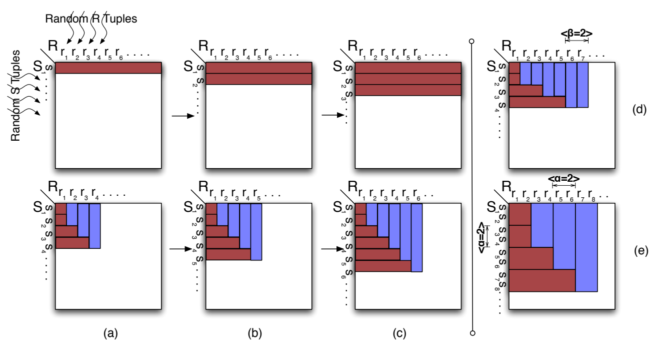

Fig. 1: (a), (b), and (c) represents a series of sampling steps. The first row depicts online nested loops-join. Whereas the second row represent a square ripple join. (d) is an example of a rectangular ripple join, where the value of $\beta_R = 2\beta_S$. (e) visualizes a *block-ripple* join where the blocking factor $\alpha = 2$.

In conclusion, scalability, interactivity, and low response time are essential to tame this exponential growth of data sizes and to support arbitrary joins which are required to facilitate interactive deep data analytics and efficient large scale computation. This paper is organized as follows. We first analyze the work done towards online aggregation and statistical estimates in sections II and III. Furthermore, we describe an efficient parallel MapReduce algorithm that supports arbitrary Θ-join conditions in section IV. Finally, in section V we describe our research plan, preliminary work towards an online parallel Θ-join system and our conclusions.

II. Ripple Joins for Online Aggregation

Traditional offline join algorithms are optimized for minimizing the overall completion time. Its drawback is that the user has to wait until the very end of the running query to evaluate an exact final answer. In contrast, online join algorithms are concerned about another factor namely, latency. More specifically, to minimize the time until an acceptable answer is provided for the running query.

A. Introduction

Ripple joins [6] represent an online family of join algorithms. In which the result is a running estimate that is continuously refreshed until it converges to the final exact answer.

The estimated proximity of the running aggregate is defined by confidence intervals generated by tools of statistical estimation theory. Ripple joins generalize the traditional blocking *block-nested-loop* and *hash*-joins to a non-blocking version. Thus, permitting updates to the running estimate of the final result. The user can control the update rate. Once set, ripple joins adaptively change their behavior according to the statistical property of the underlying data.

To use statistical estimation theory, input tuples have to be processed in random order. Such processing is similar to simply scanning data clustered randomly on disk. Otherwise, if data is clustered on a specific column, secondary random index can be used to support random ordering. Alternatively, one can scan sample base relations gotten from sampling techniques, as in [12], during processing time or materialized random relation samples.

There is a spectrum of join algorithms that rise from the tradeoff between the rate at which the running confidence intervals are updated and the degree to which the interval length decreases at each update. Offline join algorithms lie at one end of this spectrum. Online nested loops joins were previously proposed [7] and lie along this spectrum. The algorithm, as shown in fig. 1, operates as follows: at each sampling step for a join $R \bowtie S$ where $|R| < |S|$, a random tuple s is sampled from S, then R is scanned to find all the corresponding tuples that joins with s. At the end of the sampling step the running estimate is updated. Although this online algorithm is more interactive than the traditional offline version, yet, its performance is unacceptable because *a*) first, a complete scan of R is required before the estimate is updated. Thus, if it possess a large size, delays would be inevitable. *b*) the algorithm is rather rigid; as the running estimate depends on aggregate equation and the statistical properties of the underlying data. Dynamically adapting to them so as to minimize the approximation error is beneficial. In contrast, ripple joins cover the entire spectrum of algorithmic possibilities by varying the running aggregates' update rate. They are as well, designed to avoid the burden of complete relation scans and to maximize the flow of statistical information.

B. Ripple Join Algorithms

In the simplest form, known as the *"square"* version of ripple joins $R \bowtie S$, one new tuple is sampled from each of R and S at each sampling step, then joined with all the previously-seen tuples and with each other. Figure 1 visualizes how this simple ripple join swipes out the join-matrix $R \bowtie S$. As depicted in the animation, tuples are randomly sampled at the same rate. As previously described, it is desirable to adaptively vary the sampling rates to maximize statistical information and minimize the confidence intervals. This variable sampling rates would result in a *"rectangular "* version of the ripple join.

Definition 1. *The general algorithm for ripple joins retrieves β_k new random tuples, where $1 \leq k \leq K$, from the base relations $R_1, R_2, \ldots, R_K$ at each sampling step.*

For example, Figure 1 (d) depicts a *"rectangular "* version of the ripple joins with $\beta_1 = 1$ and $\beta_2 = 2$, where the second relation R retrieves twice as much as the first relation S at each sampling step. Compared to the simple "squared" ripple join, this version requires more I/O for each sampling step, thus takes more time between updates.

We next describe several variants of the ripple joins family. The traditional offline *nested-loop* joins are improved by reading from the outer relation large "blocks" of pages at a time. Similarly a *block-ripple* join can be derived along the same lines; where a new block of random tuples of R and S are retrieved and compared against each other and all the previously seen tuples. Blocking is beneficial as it amortizes the cost of rescanning one relation to join it with the other,

op	Mean:μ	Variance:σ^2
COUNT	$\mu_c = \frac{\lvert R\rvert\lvert S\rvert}{\lvert R_n\rvert\lvert S_n\rvert} \sum_{(r,s)\in R_n\times S_n} expression'(r,s)$	$\sigma_s^2 = \sigma_c^2 = \sigma^2(R)/\alpha\beta_R + \sigma^2(S)/\alpha\beta_S$, where
SUM	$\mu_s = \frac{\lvert R\rvert\lvert S\rvert}{\lvert R_n\rvert\lvert S_n\rvert} \sum_{(r,s)\in R_n\times S_n} one'(r,s)$	$\sigma^2(R) = \frac{1}{\lvert R\rvert}\sum_{r\in R}(\mu(r;R)-\mu)^2$
AVG	$\mu_a = \frac{\sum_{(r,s)\in R_n\times S_n} expression'(r,s)}{\sum_{(r,s)\in R_n\times S_n} one'(r,s)}$	$\sigma_a^2 = (\sigma_s^2 - 2\mu\rho + \mu^2\sigma_c^2)/\mu_c^2$, where $\mu = \frac{\mu_s}{\mu_c}$

TABLE I: Various mean and variance estimators, Notes: $expression'(r,s)$ and $one'(r,s)$ are equal to $expression(r,s)$ and $one(r,s)$ respectively if they satisfy the `WHERE` clause, and 0 otherwise. In case of `SUM`, for a given $r \in R$, $\mu(r;R)$ is defined as the average of $|R||S|expression'(r,s)$ for all $s \in S$, similarly for $\mu(s;S)$. Alternatively, in case of `COUNT`, $\mu'(r;R)$ is defined as the average of $|R||S|one'(r,s)$. For `AVG` estimators, $\rho = \rho(R)/\alpha\beta_R + \rho(S)/\alpha\beta_S$, where $\rho(R)$ is defined as the covariance of the pairs $\{\mu(r;R), \mu'(r;R) : r \in R\}$, and similarly $\rho(S)$ is defined.

resulting in an I/O savings factor proportional to the block size.

Another improvement to the traditional *nested-loop* join is making use of indexes; when joining $R \bowtie S$, and there is an index on the join attributes of R. Then the *Index ripple join* exploits the index to identify tuples from R that join with the randomly sampled tuple $s \in S$. This saves from much I/O costs, as full scans of the R relation are not required anymore. The drawback of such method, is that roles of "inner" and "outer" relations do no alternate, as each sampling step corresponds to a complete probe of the index on R.

Finally, we consider the *hash ripple join* variant. Where two hash tables are materialized in memory, one for R and another for S, each of them contains the tuples seen so far. Whenever a new tuple is retrieved, the hash table corresponding to the other relation is probed to find potential join results. The problem with such join is that it only supports equi-joins and it breaks down whenever memory is not sufficient to cache both relations.

C. Statistical Considerations

Ripple joins provide running estimates for multi-table queries in the form of:

Query Format
```
SELECT op(expression) FROM R1,R2,R3,..RK
WHERE predicate
GROUP BY columns;
```

where $K \geq 2$, `op` is an aggregation operator one of `COUNT`, `SUM`, `AVG`, `VARIANCE`, or `STDEV`, `expression` is an arithmetic expression that involves attributes of the base relations $R_1, R_2, R_3, \ldots, R_K$, and `predicate` is a conjunction of join and selection predicates involving these attributes.

Without loss of generality, we assume the join between two relations R and S, i.e., $K = 2$. The mean μ and variance σ^2 estimators of the various aggregation operations are summarized in Table I.

To develop tight intervals for the proposed estimators, *large sample confidence* intervals are obtained based on the central limit theorem *CLT*.

Definition 2. *The CLT for iid (*independent and identically distributed*) random variables asserts that for large n, the estimators μ_n converge to a normal distribution with mean μ and variance σ^2/n.*

We consider a standardized random variable $Z = (\mu_n - \mu)/(\sigma/\sqrt{n})$ yielding a standard normal distribution, i.e. with mean $\mu = 0$ and variance $\sigma^2 = 1$. Given $p \in (0,1)$, denote by z_p the z score (unique number) such that area under the standard normal curve between $-z_p \to z_p$ is equal to p and such that $P\{-z_p \leq Z \leq z_p\} \approx p$. Substituting with the previously defined value of Z we get $P\{\mu_n - \epsilon_n \leq \mu \leq \mu_n + \epsilon_n\} \approx p$. Thus the true value of μ lies within $\pm\epsilon_n$ with an approximate probability of p, where $\epsilon_n = z_p\sigma/\sqrt{n}$

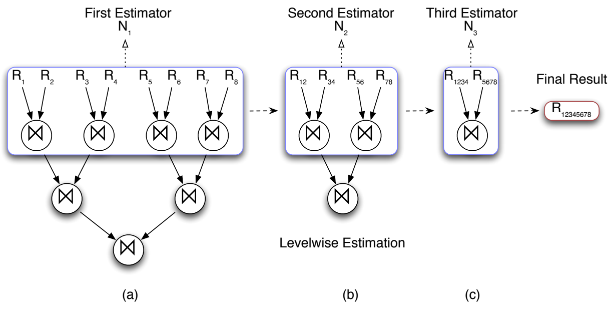

Fig. 2: Levelwise query evaluation. (a) All the bottom-level joins are evaluated concurrently in the first levelwise step, where an estimator N_1 is evaluated. (b) and (c), the second and third levelwise steps respectively where N_2 and N_3 are estimated. (d) The exact final result is emerged.

Ripple joins maximize the flow of statistical information. It does so by optimizing the values for the aspect ratios $\beta_1, \beta_2, \ldots, \beta_K$. More specifically they are assigned such that $\sigma^2 = \sum_{k=1}^{K} \frac{d(k)}{\alpha\beta_k}$ is minimized, more details in [6]

III. Scalable Approximate Query Processing with the DBO Engine

DBO [8] is a query processing engine of a prototype database system. Similar to a traditional database management system, DBO is capable of evaluating an exact answer for a query in a scalable fashion. Foremost, it can handle interactive data exploration by maintaining approximate guesses (with accuracy guarantees) for the final result at all times. To achieve this, they introduce *a*) a redesign of the traditional query processing engine to promote information sharing amongst the relational operations. *b*) a novel scheme for generating join tuples in a random manner applicable for statistical methods exploit. *c*) a generalization to previous analysis [7][6][9] for different types of queries, and derivation of unbiased estimators for queries over arbitrary number of tables.

A. Introduction

The early work of ripple joins faces scalability problems. As soon as enough data has been processed that they cannot

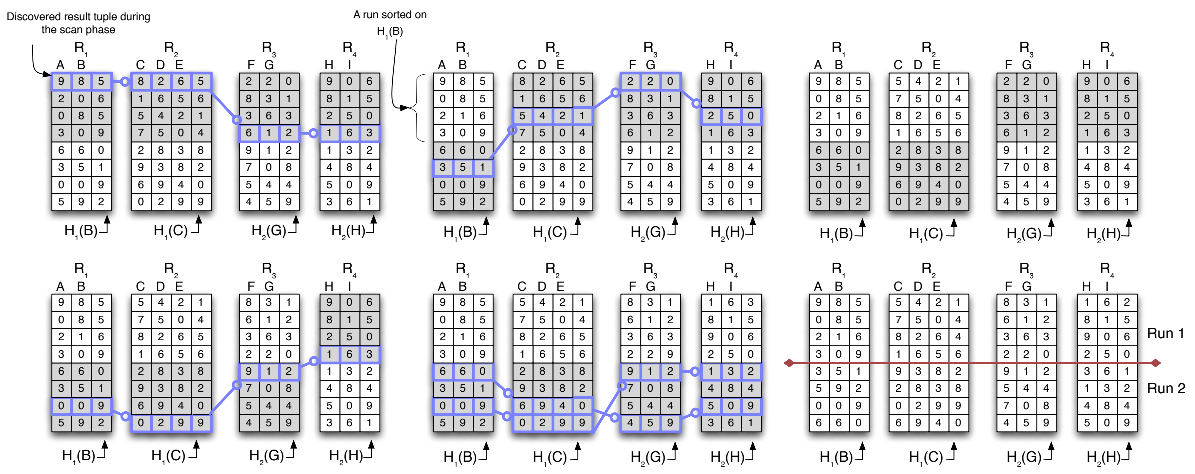

Fig. 3: An example of the scan phase. In this example, we assume the join predicate $R_1.B = R_2.C$ AND $R_2.E = R_3.F$ AND $R_3.G = R_4.H$. This first levelwise step computes $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$. In the scan phase a run from each input relation is first read into memory, joined and checked if for any immediate result tuples. Then the runs are sorted and written back to disk, taking a round-robin fashion.

be stored in main memory, the algorithm is no longer possible to proceed affectively; as it becomes necessary to page out one or more records to make room for other new ones and to page in other records to be checked against the new record. These I/O operations are random due to the randomness of the input tables, thus causing severe thrashing. Even the extended version of hash-ripple joins to a parallel environment [10] could not provide any statistical guarantees after memory is overflowed. Their intuition is that usually, the user terminates a running query early as soon as an accurate answer is provided. But this is not always the case; available memory buffers may be consumed after only a few seconds, before the approximate converges to an acceptably accurate answer. This convergence might be slow under a variety of conditions including *a*) if the join condition has high selectivity, where most of the buffered tuples contribute nothing to the final result. *b*) if the query contains a group-by clause and the group cardinalities are too many or skewed. *c*) if the database records or key values consume much memory, e.g., long character strings. DBO confronts these challenges and evaluates an approximate answer with statistical guarantees at all times until it converges to the exact final result.

B. Overview

The problem with traditional query processing engines is that relation operators are treated as *black boxes*. This makes it impossible to obtain accurate statistical estimates because intermediate results and internal state are hidden and externally invisible to the global state of the system. DBO's execution engine is quite different, as it exposes information about relational operators and intermediate results at each single level of the executing query plan. All operations at a single level of the query plan *"levelwise step"* have access to enough information to compute the final result.

As shown in fig. 2, the DBO engine proceeds by beginning with evaluating the first *levelwise step*, where each operator on this bottom level is executed concurrently. An online estimator N_1 is computed and as time progresses the estimator achieves more accuracy until it freezes as the step completes. The resulting intermediate results are used as input to the next *levelwise step*. Similarly an estimator N_2 is evaluated for this level at all times. The estimator N_2 is combined with N_1 to produce an overall single estimate. The same procedure is applied at each subsequent level until the last *levelwise* step. At each *levelwise* step, an individual join operator is implemented as a modified version of *sort-merge join*. This join algorithm can be described in two phases; a scan phase and a merge phase.

C. Scan Phase

The *scan phase* is analogous to the sort phase of a *sort-merge join* but with several variations such as *a*) the immediate discovery of output tuples by joining subsets of tuples stored in the memory in a manner similar to the ripple join. These output tuples are used to generate an estimate for the final result. *b*) the randomized order of input data to provide statistical guarantees for estimators. Since the input of an operator might be pipelined from the previous one, then the intermediate output results also have to be random.

The scan phase proceeds as follows:

1) At the beginning of the scan phase, one run from each relation (or intermediate output) is read into memory and joined in search for output tuples that are used to obtain an unbiased approximate for the final result.
2) Then, the run from the *first* relation is sorted and written back to disk. Rather than sorting the run over the join key $R_i.Key$, it is sorted on the value of $H(id, R_i.key)$, where H is a hash function and id is an input seed value to the function. In order to ensure that there is no correlation amongst the different joins, a different value for the seed id is used for each join. The randomized hash function ensures a sort order based on a randomized lexicographical order that is statistically independent of the tuples' attributes. This randomized order facilitates

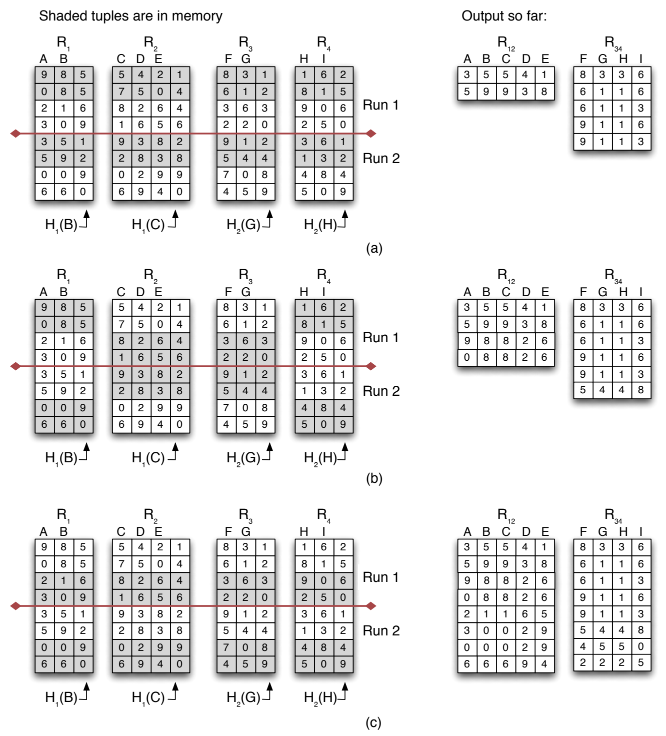

Fig. 4: The merge phase of a levelwise step used to compute $R_{12} \leftarrow R_1 \bowtie R_2$ and $R_{34} \leftarrow R_3 \bowtie R_4$ where $R_1.B = R_2.C$ AND $R_2.E = R_3.F$ AND $R_3.G = R_4.H$. First, the head of each run produced by the scan phase is read into memory and joined then it proceeds as a merge-join.

the direct use of output tuples into the next *levelwise* step.

3) After the run is sorted and written back to disk, another set of records is paged into the memory. The choice of paged-in runs adopts a *systematic round robin fashion*, where the second run of the first relation R_1 is read in and joined with all the other tuples in memory. Then the second run from the second relation R_2 and the cycle begins again. An example is depicted in fig 3.

D. Merge phase

The *merge phase* is analogous to the merge phase of a traditional *sort-merge join* except that all merges are run concurrently; the head of each run of each input relation in a *levelwise* step is brought into memory, and runs of records from each output relation are produced in a round-robin manner in order to pipeline them into the scan phase of the next *levelwise* step without writing them back to disk. An example is depicted in fig 4.

E. Statistical Considerations

The previous sections describes at a high level how the join algorithm proceeds. In this section we investigate how online estimates are computed in the DBO system. As described previously, there is an estimator N_i for the query result at each *levelwise* step. These estimators are combined at all times to provide an accurate unbiased statistical estimator for the final result as $N = \sum_{i=1}^{d+1} w_i N_i$ and variance $\sigma^2(N) = \sum_{i=1}^{d+1} w_i^2 \sigma^2(N_i)$. In order to minimize the error associated with the estimator N, we seek to minimize the variance of N over all possible weights. Using Lagrangian multiplers, we can minimize σ^2 by choosing $w_i = 1/\{\sigma^2(N_i)\sum_{j=1}^{d+1}\frac{1}{\sigma^2(N_j)}\}$. Once these values are in hand, it is easy to compute the confidence bounds using the *CLT* as previously described in section II-C.

Given that $T(i,j,k) = R_{j,i} \times R_{j+1,i} \times \ldots \times R_{k,i}$ which represents the cross product of all of the tuples in the i^{th} run of input relations j through k, after r runs have been processed from each relation in a single *levelwise* step the following is equal to the sum of aggregate function f over all tuples that have been discovered:

$$\alpha = \sum_{a=1}^{r}[\sum_{t\in T(a,1,n)} f(t)] + \sum_{a=2}^{r}\sum_{b=1}^{n-1}[\sum_{t\in T(a,1,b)}[\sum_{t\in T(a-1,b+1,n)} f(t_1.t_2)]]$$

In the first levelwise step N_1 an estimator to the final result is computed by scaling the aggregate α with a scaling factor β, where β is the ratio of the size of the overall data space to the number of tuples discovered by the scan phase.

$$\beta = \frac{|R_1 \times R_2 \times \ldots \times R_n|}{\sum_{a=1}^{r}|T(a,1,n)| + \sum_{a=2}^{r}\sum_{b=1}^{n-1}|T(a,1,b)||T(a-1,b+1,n)|}$$

The estimation process in the subsequent *levelwise* steps differs from the first one for two reasons. First, the intermediate output tuples which are produced by the merge phase are in a semi-random order where tuples with the same join key are generated all in one group. Second, knowledge about the cardinalities of the intermediate relations which are required for the estimation process are unknown apriori. Because the results of the merge phase in the i^{th} *levelwise* step are pipelined to the scan phase of the next $(i+1)^{th}$ *levelwise* step. To handle the first problem, DBO samples blocks rather than tuples; where a group of tuples that have the same join key are viewed as a single indivisible output tuple. Thus any correlation due to grouping is removed. Whereas for the second challenge, DBO partitions the output space into p approximately equi-sized ranges of key values. Therefore a clump of output tuples has a probability of $1/p$ of falling into a given run yielding a Bernoulli sampling. Thus an estimate X produced by a join between the in-memory runs can be scaled by p^n to obtain an unbiased estimate for the eventual query result. Given there are $1 + (r-1)n$ discovered estimates X, the overall estimate N_i is computed by scaling the aggregate α with a scaling factor of

$$\beta = \frac{p^n}{1 + (r-1)n}$$

For more information about derivations, proofs, and variance estimation, the interested reader might refer to [8] for more details.

IV. Processing Θ-Joins Using MapReduce

The very large data sizes pose many challenges for data analysis. To ensure reasonable response time for such analysis, parallel computation is essential. MapReduce [5] has emerged as a very popular parallel processing paradigm. Although the MapReduce paradigm does not directly support joins, there has been some progress on implementing equi-join algorithms in MapReduce [1], [2], [13], [14], [15]. Equi-joins are basically implemented by exploiting MapReduce's key-equality based data flow management. Nevertheless, the need for load balancing and supporting arbitrary complex join conditions has evolved, e.g., for spatial data, band-joins and spatial-joins are common. Inequality and similarity joins are important for correlation analysis as well.

[11] proposes a randomized algorithm, called the 1-Bucket-Θ, to implement any arbitrary Θ-join on the MapReduce framework which only requires the minimal statistic of relations cardinality. They also propose algorithms that improve on 1-Bucket-Θ for a certain class of join conditions as well as with sufficient required statistics knowledge.

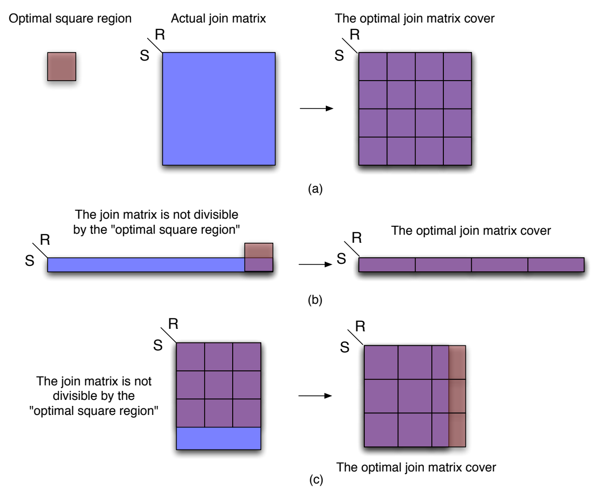
Optimal square region
Actual join matrix
The optimal join matrix cover
R
S
(a)
The join matrix is not divisible by the "optimal square region"
The optimal join matrix cover
(b)
The join matrix is not divisible by the "optimal square region"
The optimal join matrix cover
(c)

Fig. 5: Matrix-to-reducer mappings in the various cases. (a) The first case, where R and S dimensions are multiples of $\sqrt{|R||S|/r}$. (b) and (c) represent the cases where the join matrix is indivisible by optimal square region, where in (b) $|R| < \frac{|S|}{r} < |S|$ and in (c) $\frac{|S|}{r} < |R| < |S|$

A. Overview of Joins and MapReduce

The standard equi-join implementations on MapReduce all revolve around the same logic of making the join attribute the key, to ensure that all tuples with identical join attribute values are processed altogether in one invocation of the *reduce* function. This basic algorithm suffers from two problems. First, the number of reducers are limited to the number of distinct values in the join attribute, which implicitly restricts parallelism. Second, this algorithm has no sense of load balancing; when the join is applied over skewed data (which is always the case) some reducers would perform more work compared to others, having variances in the distributed load amongst the workers. Thus delaying the completion of the job.

Over and above, it is not clear how to support the different join conditions. For example consider a join between datasets R and S with an inequality condition, i.e., $R.A \leq S.A$. Such join, seems to be difficult to be implemented over MapReduce because each tuple of S not only has to be joined with R tuples of the same A value but also with those of smaller values. Nevertheless, this proposed solution suffers from many drawbacks; *a*) The join algorithm is dependent upon the data, as it generates a large number of duplicates relying on the join attribute values. *b*) If the attribute A is not an integer, or can have negative values, it is impossible to enumerate all the values smaller than a given value $S.A$.

To support any Θ-join condition and overcome all of the previously mentioned problems, the authors model a join between two relations as a *join-matrix*. Since any Θ-join is a subset of the cross-product, the matrix can represent any join condition.

B. The 1-Bucket-Θ Algorithm

Each cell in the *join-matrix* represents a potential join output. The idea is to assign each cell to a specific reducer which is responsible for evaluating the output of this cell, i.e., yielding either a true or false value. Each cell is assigned to exactly one reducer so as to avoid expensive post-processing or duplicate elimination. There can be many possible mappings that cover the matrix cell. Hence, the problem can be formulated as given r reducers, what is the mapping from the *join-matrix* M cells to reducers R so as to minimize job completion time.

To minimize job completion time, we need to minimize both *reducer-input* and *reducer-output* related costs. We say that a join problem is *input-size dominated*, if reducer-related costs dominate job completion time. If reducer-output related costs dominate job completion time, then the join problem is termed as *output-size dominated*. The join problem category is dependent upon the join condition and algorithm. To reduce the completion time of *output-size dominated* and *input-size dominated* queries, we need to minimize **max-reducer-output** and **max-reducer-input** respectively.

The 1-Bucket-Θ algorithm is devised to minimize both output and input based costs as it is based upon the following lemma.

Lemma IV.1. *A reducer that is assigned to c cells of the join matrix M will receive at least $2\sqrt{c}$ input tuples*

1) Cross-Product: In the cross-product case, the following lemma describes the lower bounds that can be achieved.

Lemma IV.2. *In case of the cross-product $R \times S$, given r reducers, the lower bound for any matrix-to-reducer mapping to max-reducer-output is $|R||S|/r$, and to the max-reducer-input is $2\sqrt{|R||S|/r}$*

These lower bounds can be achieved if the cardinalities of both $|R|$ and $|S|$ are multiples of $\sqrt{\frac{|R||S|}{r}}$, otherwise the problem of minimizing max-reducer-input for a given value of max-reducer-output can be formulated as an integer linear programming problem which are known to be NP-hard. Thus the authors propose a mapping mechanism that is close to optimal, visualized in fig. 5, and summarized as follows:

1) In case of $|R|$ and $|S|$ are multiples of $\sqrt{|R||S|/r}$, where $|R| = c_R\sqrt{|R||S|/r}$ and $|S| = c_S\sqrt{|R||S|/r}$ for integers $c_S, cT > 0$. Under these conditions the optimal mapping would be a join-matrix partitioned into c_S by cT squares of size $\sqrt{|R||S|/r}$ by $\sqrt{|R||S|/r}$. Such mapping would achieve the lower bounds mentioned in the previous lemma.
2) In case of $|R| < \frac{|S|}{r} < |S|$, the max-reducer-input is minimized by partitioning the matrix into a single row of r rectangles of sizes R by $\frac{|S|}{r}$.
3) In case of $\frac{|S|}{r} < |R| < |S|$, let $c_R = \lfloor |R|/\sqrt{|R||S|/r} \rfloor$, $c_S = \lfloor |S|/\sqrt{|R||S|/r} \rfloor$. Where c_R and c_S indicate how many optimal squares of side length $\sqrt{|R||S|/r}$. To cover the remaining cells that might have been not covered by these optimal squares the heights and widths of these squares are being scaled by a factor of $(1 + 1/min\{c_R, c_S\})$. This ensures an upper bound for max-reducer-output is $4|R||S|/r$, and a bound of $\sqrt{|R||S|/r}$ for max-reducer-input.

After having the mapping setup, converting it into a MapReduce program is conceptually straightforward. Whenever a new R tuple is received, the map function finds all the regions intersecting the row that represents this tuple in the join-matrix and generates a pair of region key and the tuple. The same concept is applied for any incoming S tuple where the map function similarly finds all the intersecting columns rather than rows. The problem now only lies on how to identify the exact row (or column) in the matrix for an incoming R (S) tuple. As there is no deterministic mechanism for the *map* function to know how many row (column) tuples have been processed before this current one being computed. To overcome this and guarantee that tuples are assigned to the appropriate rows and columns, an additional MapReduce preprocessing step is run that assigns unique row and column numbers for R and S respectively.

To overcome this additional MapReduce step, a randomized algorithm is proposed; such that whenever an incoming R (S) tuple has been received, the map function randomly chooses a row (column) in the join-matrix M. Affectively, it creates an output tuple for each region that intersects with this row (column). Notice that with randomization, there are some rows and columns that are randomly selected multiple times for different tuples, others are not chosen at all. But this variation is very unlikely when the number of tuples being processed is large.

2) Θ-Joins: Consider an arbitrary join condition with selectivity σ. To minimize max-reducer-output, each reducer has to affectively produce $\sigma|R||S|/r$ tuples. Due to the randomization effect of the 1-Bucket-Θ algorithm, which assigns random samples of R and S among the workers, the join output is averaged out between all the reducers. Thus any algorithm that might compete with the 1-Bucket-Θ algorithm would perform better from the perspective of input-related costs.

Lemma IV.3. *Given that* $0 < \delta \leq 1$. *Any join-matrix to reducer mapping that covers at least* $\delta|R||S|$ *of the whole matrix* $|R||S|$, *has a max-reducer-input of at least* $2\sqrt{\delta|R||S|/r}$. *Comparing this to the worst case of the 1-Bucket-Θ algorithm we get* $\frac{4\sqrt{|R||S|/r}}{2\sqrt{\delta|R||S|/r}} = \frac{2}{\sqrt{\delta}}$.

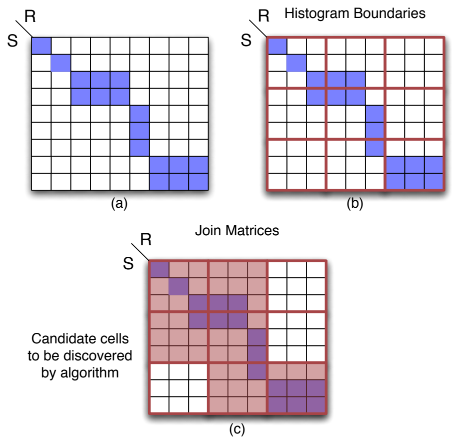

Fig. 6: (a) represents the actualjoin matrix, where the shaded portions depict result cells. In (b), histograms are created and neglected regions are discovered as shown in (c).

As evident from the previous lemma, we conclude that unless δ is too small, i.e., very selective join conditions , there is no other matrix-to-reducer mapping that will result in a much lower max-reducer-input compared to the 1-Bucket-Θ algorithm. Nevertheless, in practice it is hard to find such efficient mapping for various reasons; *a*) First, more input statistics, that are beyond the cardinalities of $|R|$ and $|S|$, ought to be known to identify regions that need not be covered. *b*) Second, the join condition might be an arbitrary user-defined functions. Thus, to know which regions to bypass requires a preprocessing stage beforehand, but this defeats the purpose of the join from the first place. Thus for selective join conditions, there might be more efficient algorithms than the 1-Bucket-Θ algorithm, which would neglect many portions of the join-matrix and not assign them to any reducer. But in practice it is quite hard to find them due to unavailable statistics and complex user defined join conditions.

3) Exploiting Statistics: For a popular class of join conditions such as band, inequality, and similarity joins, we can further exploit statistics to identify regions from within the join-matrix that do not contain output tuples. This is achieved by computing an *approximate equi-depth histogram* on both relations R and S through two MapReduce jobs. In the first MapReduce job, n records are sampled from each relation R and S by outputting a tuple with probability $n/|R|$ or $n/|S|$ respectively otherwise, discarding it. Then these n records are sorted by the join attribute and grouped by the reducer to compute *approximate k-quantiles*. The second MapReduce job, passes by each dataset and counts the number of tuples that fall into each bucket.

Having the *approximate equi-depth histogram* in hand, now it is possible to neglect those regions in the join-matrix which we are confident that no output could extracted from them as explained in 6.

V. Preliminary Results And Research Proposal

In this section we present *a*) our research plan in V-A *b*) our preliminary results on Online Θ-joins and our current work in V-B, and *c*) our conclusions in V-C

A. *Future Directions*

We have analyzed previous work towards early results and online aggregation. This work suffers from two limitations; first the join algorithms are not scalable as parallelization has become a prominent solution to face the rapid growth of data. Second, the join algorithms are restricted to equi-joins whereas sophisticated data analysis requires the support for arbitrary join conditions. Θ-joins, on the contrary, provide an elegant solution to process any join condition in a scalable parallel fashion, but are designed for batch processing systems.

We plan to combine the advantages of both solutions; the early results (low latency) of online aggregation methods alongside with the generalization and scalability (high throughput) of the Θ-joins algorithms. However, designing a scalable online Θ-join system faces many challenges, some of which are *a*) No previous or minimal knowledge of statistics for relations are known, even the bare minimum requirement of [11], relation cardinalities. Even if the cardinalities of the base relations are known, those of intermediate relations (in multi-level join operations) cannot be anticipated beforehand. *b*) Coping with the first challenge by changing the matrix-to-reduce-mapping during execution gives rise to another challenge, namely, designing a dynamic model to efficiently use the available resources to decide and execute these changes. *c*) Data arrival rates suffer from data bursts or fluctuations. The right balance of "stability" has to be found. A very unstable system has excessive decision overheads while a too rigid system does not adapt well to the data dynamics. *d*) It is not clear yet how to support statistical guarantees and confidence intervals in such a parallel system.

B. Cyclone: *Online Θ-joins*

We confront the first three challenges by introducing a dynamic model for on-the-fly relation statistics, a simple, effective and efficient extension to the Θ-joins algorithm to operate in a fully online manner. The model is used to predict quantities used in the join algorithm and is designed to adapt to data fluctuations. This is principally used to predict relation cardinalities.

Furthermore, the model decides for assignment changes and dynamically adapts to highly utilize the given resources. As the choice of an efficient matrix-to-reducer mapping results in better data distribution, less redundant tuples, less communication cost, and an overall better utilization of memory and storage.

We evaluate our model, over which we built Cyclone, a parallel online processing system prototype for Θ-joins. Our preliminary experimental results show that Cyclone efficiently utilizes resources and is able to dynamically adapt to fluctuations in data arrival rates. Most importantly, it is optimized to achieve high throughput by scaling out, and acquire low latencies in producing early online results.

C. *Conclusions*

High throughput and low latency in providing early results are essential to tame the explosive growth of data in analytical workloads. We showed how online aggregation provides early approximate results with statistical guarantees. Furthermore, we examined how arbitrary Θ-joins can be supported using the scalable MapReduce framework. Then, to achieve the best of both worlds, we analyze the challenges facing the combination of both solutions. Finally, we briefly discuss our preliminary work and early results.

References

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[2] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[3] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *Proc. VLDB Endow.*, 2(2):1481–1492, Aug. 2009.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

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

[6] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Conference*, pages 287–298, 1999.

[7] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. SIGMOD Conference*, pages 171–182, 1997.

[8] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4), 2008.

[9] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 563–574, New York, NY, USA, 2005. ACM.

[10] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 252–262, New York, NY, USA, 2002. ACM.

[11] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.

[12] F. Olken. Random sampling from databases, 1993. PhD Thesis, UC Berkeley.

[13] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 267–273, Berkeley, CA, USA, 2008. USENIX Association.

[14] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[15] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.