Query Optimization for Streaming Data

Nipun Mittal
193050005

June 9, 2020

Seminar Report (Spring '20)

Guide: Professor S. Sudarshan

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

Acknowledgment

I would like to extend my gratitude to Professor S. Sudarshan who guided my throughout this semester to pick seminal research papers relevant to my domain, helped me clear my doubts whenever needed and provided his valuable feedback on this report.
I also thank my colleagues who helped me develop better understanding by various discussions.

Contents

List of Figures

Chapter 1

Introduction

Today, data analytics mines huge distributed datasets and process them for business or academic usecases. Querying these massive datasets can take upto hours if not done efficiently. So these is a great need of good optimization techniques which can develop an execution plan to query efficiently. There has been many techniques proposed targeting specific set of usecases. This report presents these different optimization techniques and analyze their performance gain on the traditional techniques.

The report is divided into five chapters. chapter 2 presents the query optimization techniques for traditional database engines typically operating on a single machine or a cluster of machines in a rack. It describes the traditional Volcano optimizer [12] and the extensions proposed by Prasan Roy in his PhD thesis [1] and in the paper by Roy and Sudarshan [2] for multi-query optimization. chapter 3 shows the SCOPE [3] declarative language developed at Microsoft aiming to decouple the optimization tasks from the developer to increase their productivity and also to avoid sub-optimal execution plans made by a developer. SCOPE claims to have the first framework which incorporates partitioning along with grouping and sorting into its optimizer engine [4]. It compares the execution time of the query plan generated by considering partitioning after the plan generation and during the plan generation. chapter 4 shows optimization techniques which leverages recurring nature of periodic queries for massive distributed databases. It gathers the statistics of the running query and store it to optimize the re-run of the same query (or query fragment) [5]. This chapter also presents the continuous evolution of execution plan of a query as the query executes [6]. This technique is useful for massive distributed queries which typically executes from minutes to hours and optimizing these queries as it executes can potentially reduce the execution time by adjusting the query plans on the basis of current gathered statistics. Finally, chapter 5 presents the optimization techniques for the query processing on streaming data. Stream data is different from the batch data as it comes from the unbounded stream of data and its timely nature is also important. It describes Apache Flink [7], a unified model for batch processing and stream-data processing with a single execution engine and different API(s) for these two paradigms. Flink shows that batch data can be treated as a special case of stream-data (with some additional optimizations tailored for batch systems) whose execution can be incorporated with the stream-processing engine. The chapter then describes the work of Bartnik [9], which proposes a dynamic model for stream-processing engines. It shows that traditionally the configuration of the stream-processing systems can't be changed without stopping the query. It proposes different protocols to adjust system resources based on the query workload, adding and updating the operator instances of the query. This

chapter then presents Unblocking Checkpointing mechanism [10] for iterative data processing. It compares the traditional blocking checkpointing mechanism (where system stops during checkpointing) with the proposed unblocking mechanisms which can checkpoint data without pausing the data processing with consistency guarantees. It shows unblocking checkpointing clearly performs better than the blocking ones. The chapter finally presents NebulaStream Platform [8] for IoT data processing and proposes many solutions to scale the system with the ever increasing IoT data and its compute.

Chapter 2

Query Optimization in Traditional Database Engines

This chapter first explains the traditional query optimization techniques and extend them to exploit optimization among multiple queries simultaneously.

2.1 Traditional Query Optimization

Traditionally OLTP queries generally involve few relations and simple predicates which can be easily executed without much need of query optimization. However as decision support query engines are coming up, they include complex and sophisticated queries which if naively executed can result in high performance bottleneck. This arises the need of better query optimizers which results into two main optimizer types: SystemR [13](Bottom-up paradigm) and Volcano [12](Top-down paradigm). This paper[1] uses Volcano to implement multi-query optimization algorithms for multiple reasons including it doesn't depend on data/execution model. For optimization it uses the cost models to compare different alternate plans. The cost of an operator is computed by the characteristics of input relations including their number, size of each relation and distribution of values of the relevant attributes in each relation.

2.1.1 Volcano Optimizer Design

Steps to find best plan:

- Generate all Logical equivalent query plans using logical transformation like changing join order, predicate pushdown etc and create a Logical Query DAG by combining all plans.
- For each logical plan, generate all possible physical plans which determines how to implement that plan. For join operator, hash-join or sort-merge join could be generated as physical plan. Again combine all physical plans to create Physical Query DAG.
- Finally enumerate over all physical plans, compute cost of each plan and select the least cost plan.

Logical Query DAG

Logical Query DAG is a Directed Acyclic Graph with operator nodes and equivalence nodes. In the Figure 2.1 operator nodes are represented in circle and equivalence nodes in square. An operator node represents the operation like join/select. Equivalence node represents the class of logical expressions which generates the same result set, all of its child nodes represents it. LQDAG can represent multiple queries in combined form. The algorithm to generate the LQDAG is given in the thesis [1] which has many subtle tricks to reduce the search space.

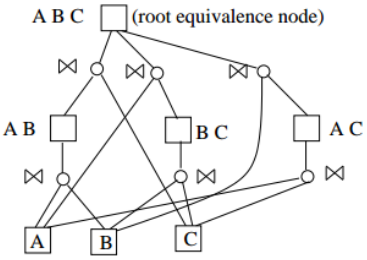


Figure 2.1: Example of a logical DAG for $A \bowtie B \bowtie C$ [1]

Physical Query DAG

LQDAG only represents the abstract semantic aspect of the query like the order in which relations have to be joined but doesn't tell the exact algorithm to do so. It also doesn't deal with the physical properties of the node like sort order and treat them as same equivalence node. However, physical properties are important as they affect the execution cost of an algorithm. For instance, merge-sort execution cost will be different depending on whether the given relations are sorted or not. Physical Query DAG explicitly deals with physical properties. The Figure 2.2 depicts the Physical Query DAG of the query. The details of the algorithm is omitted for the obvious reasons.

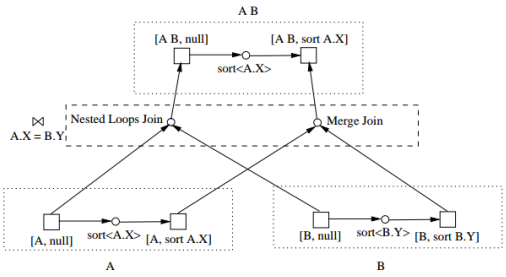


Figure 2.2: Example of a physical query DAG for $A \bowtie B$ also involving physical properties of the nodes [1]

2.2 Multi Query Optimization

This section presents the work of Roy and Sudarshan [2] which talks about the need of Multi-Query optimization and also propose some novel optimizations to speedup the execution time of queries. Today in enterprise environment we see lots of complex queries which generally involves many common sub-expressions like i) parametrized queries have many invocations with most of the invariant parameters, ii) in big data analytics, queries were released in a batch which opens the possibility of common sub-expressions, iii) in nested queries, multiple invocations of the inner query have many overlapping expressions and iv) views are extensively used which could be referred multiple times hence opens the possibility of materialization of views which could reduce the execution time.

Traditional query optimizers optimize each query at a local level which could easily lead to sub-optimal execution plans.

Consider the following example from the paper [2]. Let there are two queries Q_1 and Q_2 whose locally optimal plans are $(R \bowtie S) \bowtie P$ and $(R \bowtie T) \bowtie S$ respectively. However, they share a common sub-expression $R \bowtie S$ and if somehow Q_2 knows that Q_1 has already computed

$R \bowtie S$ then $(R \bowtie S) \bowtie T$ could be more optimal than $(R \bowtie T) \bowtie S$. So, this depicts the need of global query optimization which generates execution plan for a bunch of queries.

By taking the motivation from the above mentioned use-cases, the authors proposes many optimization techniques. After generating the physical AND-OR-DAG(explained in the previous section) for each query, they consolidate all the queries using a pseudo operation root node with all the root equivalence nodes of the queries as its children. They have also taken subsumption into account like we can compute $\sigma_{A<5}(E)$ can be computed from $\sigma_{A<10}(E)$ as computing $\sigma_{A<5}(E)$ can be costlier in compare with computing $\sigma_{A<5}(E')$ where $E' = \sigma_{A<10}(E)$ and E' is available.

2.2.1 Traditional Volcano Optimization Algorithm

The optimizer operates on extended DAG and compute the cost of each node in DFS(depth first search) manner. The cost model is:

$cost(o) = cost\ of\ executing\ (o) + \sum_{e_i \in children(o)} cost(e_i)$

$cost(e) = min\{cost(o_i) | o_i \in children(e)\}$

Here, o represents operation node which need all its children equivalence nodes hence *sum* and e represents equivalence node which can use any of its children operation nodes hence *min*.

Authors proposes three different novel algorithms by modifying the existing algorithm, Volcano-SH, Volcano-RU and Greedy algorithm.

2.2.2 Volcano-SH Algorithm

Volcano-SH exploits the common-subexpressions among multiple queries. Suppose if a sub-expression is used by multiple queries then we can materialize its result so that another query can just read it without computing it. However, if materialization cost is high then it might be better to recompute it. So, the decision to materialize a node e is based on the comparison that if *cost(e) + matcost(e) + reusecost(e) * (numuses(e) - 1) < numuses(e) * cost(e).* Here, matcost(e) is the cost to materialize the node e, reusecost(e) is the cost of use e after it is materialized and numuses(e) is number of times e has been used. The problem with the above formula is that numuses(e) and cost(e) depends on which other nodes have been materialized. This problem is solved by traversing the DAG in bottom-up manner and decide the cost(e) based on which of its successor nodes have materialized. However, for numuses(e) we need the materialization information of the ancestors of e which isn't possible. So, a heuristic has been used by computing numuses(e) as the number of parent nodes of e.

2.2.3 Volcano-RU Algorithm

Consider Q_1 and Q_2 from the example in the section 2.2. If we realize that $R \bowtie S$ has been used by Q_1 then we can consider this in computing the plan for Q_2 and resulting plan would be $(R \bowtie S) \bowtie T$. In Volcano-RU, given a batch of queries, the queries are executed in sequential order and the optimizer stores all the sub-expressions in the plan selected by previous queries and consider it while computing execution plan for the current query. In addition to this, optimizer also can choose to materialize any node if it's used in previous query and it can reduce cost. After this, Volcano-SH is being executed to further uncover and exploit common sub-expressions.

2.2.4 Greedy Algorithm

As we can see from the above algorithms, the best plan tends to find optimal subset of nodes in the query DAG which when materialized results in the minimum cost execution plan. In greedy algorithm, we consider a subset of nodes to be materialized and compute the execution plan based on that. Now, if we consider every subset of nodes then the complexity will be very high and make it unfeasible, so multiple heuristics have been used to make it feasible.

```
Procedure GREEDY
Input: Expanded DAG for the consolidated input query Q
Output: Set of nodes to materialize and the corresp. best plan
    X = φ
    Y = set of equivalence nodes in the DAG
    while (Y ≠ φ)
L1:     Pick x ∈ Y which minimizes bestcost(Q, {x} ∪ X)
        if (bestcost(Q, {x} ∪ X) < bestcost(Q, X) )
            Y = Y - x;   X = X ∪ {x}
        else Y = φ
    return X
```

Figure 2.3: The Greedy Algorithm [2]

In Figure 2.3, the algorithm picks one node after each iteration which gives maximum reduction in the existing cost and materialize it. $bestcost(Q, X)$ represents the execution cost of query Q given X subset of nodes being materialized. Now, there are few optimizations that has been done to make this algorithm more feasible.

Sharability

The algorithm only consider nodes in the Y set of initial nodes which have been refereed more than one time as it make sense to not materialize node which is used exactly once in the DAG. It can easily compute this in the order of the size of the DAG using memoization in bottom-up manner.

Incremental Cost Update

As we can see that in line $L1$ of the algorithm there will be multiple calls to $bestcost()$ function with different node. However, in each call there is only a small change in the subset of nodes i.e, one node is deleted and one new node is added. So, $bestcost()$ can be incrementally computed instead of executing it from scratch for each call. The algorithm updates the cost of the node being changed and propagate the change through the ancestors to all the way to the root. Since, there is a chance that a change will be propagated through a node multiple times, we can sort the DAG nodes in topological order and update them in that order.

The Monotonicity Heuristic

Let's define $benefit(x, X)$ as $bestcost(Q, X) - bestcost(Q, \{x\} \cup X)$. We can safely assume that if we add more nodes to the subset X, $benefit(x, X)$ will not increase i.e, $\forall X \subseteq Y, benefit(x, X) \geq benefit(x, Y)$.

The algorithm leverages this, by avoiding the computation of $bestcost(Q, \{x\} \cup X)$ after each iteration. After one iteration it has calculated the $benefit(Q, \{x\})$ for each node. In second iteration instead of re-computation, it maintains a heap of costs of every node created in first iteration and select the second best from it in second iteration and so on. It picks the best benefit node from the heap, recompute its benefit and if it's still be the best, the algorithm materializes that node otherwise it inserts it back and pick the best again from the heap.

Apart from these optimizations, authors also incorporated index creation, exploited sub-expressions in nested queries and optimize multiple invocations of the parametrized queries. Based on the TPC-D benchmark queries, these optimizations improved the execution time from a factor of 1.2x to 2x.

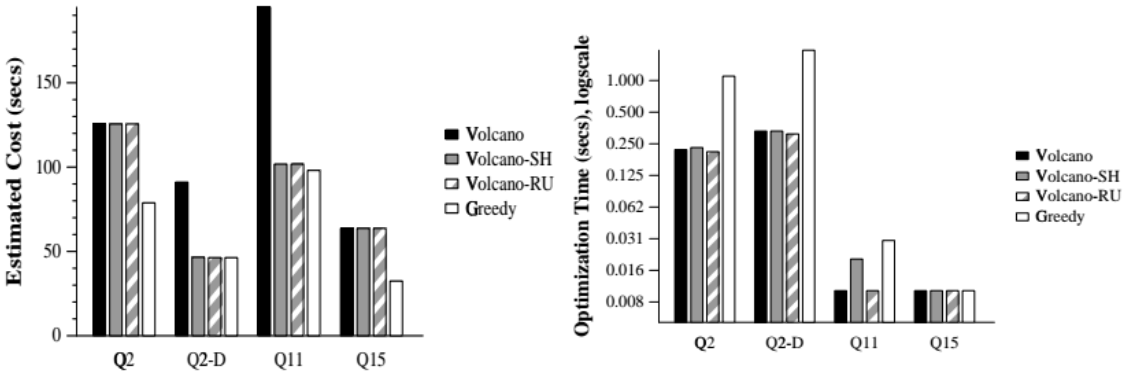


Figure 2.4: Optimization of Stand-alone TCPD queries [2]

Figure 2.4 shows one of the performance charts showed in the paper [2] comparing the estimated cost and optimization time of different algorithms. The results clearly shows the consistent performance gain from Greedy Algorithm and significant gains from Volcano-SH and Volcano-RU in some queries. The performance study also consider batch and scale-up queries and shows consistent gains. They have tested their queries on Microsoft SQL-server. The paper concluded that multi-query optimization can be practically used to significantly improve the execution time of queries.

Chapter 3

SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets

Today, data processing of massive datasets requires new programming paradigms and languages which can parallelize the processing and efficiently compute the results. Due to this, many programming models have been developed including popular Map-Reduce model which maps the input on different machines, perform computation parallelly and reduce the result into single/multiple machine(s). However, this model has its own limitations. Programmer has to convert its program to comply with map-reduce model which is not possible/laborious for some tasks. Such conversion can be error-prone also. Besides this, since user is implementing map-reduce functions user should decide the order of execution of the tasks which can lead to sub-optimal execution plan and significantly degrade the performance.

Due to above mentioned reasons, the author of this paper [3] have developed a programming language SCOPE (Structured Computations Optimized for Parallel Execution). SCOPE is a declarative language which is syntactically same as of SQL (this is done intentionally to decrease the learning curve). SCOPE doesn't require programmer to define the order of execution for the task. It is same as SQL where user tells what to do and the compiler and optimizer decides the method and parallelism to perform the given task. SCOPE is highly extensible. Users can define their own functions and override given operators: extractors (parsing and constructing rows from a file), processors (row-wise processing), reducers (group-wise processing), and combiners (combining rows from two inputs). SCOPE also supports views like SQL which can be parametrized also.

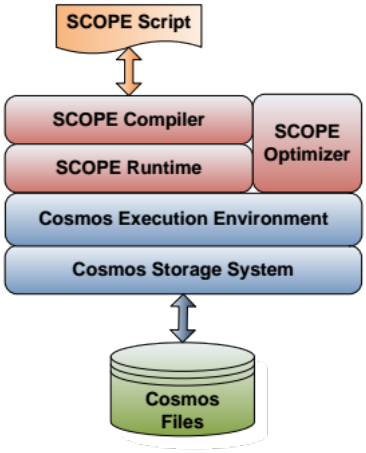


Figure 3.1: Cosmos Platform layers [3]

Figure 3.1 shows the different layers of the Cosmos environment. Cosmos Storage is a append-only file-system which is massively distributed with high availability and reliability. Cosmos Execution Environment defines the execution plan of the given SCOPE query. It creates a DAG with nodes as programs and edges as data flow. It decides the scheduling of the execution of each node in highly parallel environment.

SCOPE provides all the main features of SQL like Select, Join and UDFs. Besides this, SCOPE has three main commands:

- **Process:** Process takes a rowset as input and for each row it outputs any number of rows

(including zero) as output.

- **Reduce:** Reduce takes a rowset as input, group the rows on the grouping column(s) and for each group it outputs any number of rows (including zero) as output.
- **Combine:** Combine takes two row sets as input and output a rowset by combining the input in some way.

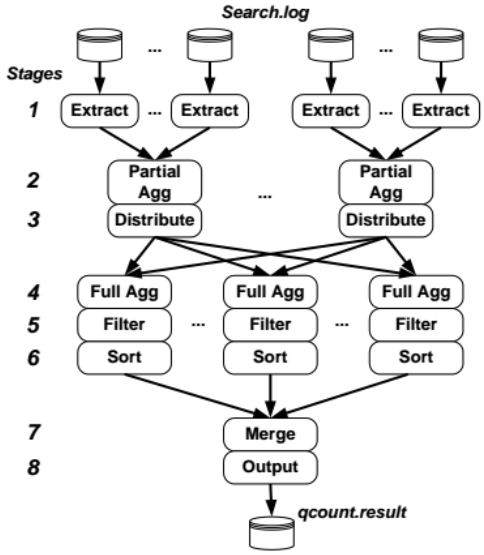
There are many features of the SCOPE described in the paper, are omitted from this report.

Example Query [3]: A QCount query compute frequencies of the queries executed on the database and returns the queries executed more than 1000 times in decreasing order.

```
SELECT query, COUNT(*) AS count
FROM "search.log" USING LogExtractor
GROUP BY query
HAVING count > 1000
ORDER BY count DESC;

OUTPUT TO "qcount.result";
```

(a)



(b)

Figure 3.2: (a) SCOPE QCount query script, (b) Execution of QCount query [3]

Figure 3.2 depicts QCount query and its execution plan generated by SCOPE optimizer. In Stage 2, partial aggregation has been done on query column by machines running in the same rack. In Stage 3, partition the result on grouping column "query" and distribute it. In Stage 5 after full aggregation, filter out rows with count ≤ 1000. Thereafter, sort the result in parallel and merge the row sets into single relation.

Experimental Evaluation: In Experimental Evaluation, Authors shows that the query performs scales linearly with the number of machines involved in the execution (cluster size) and with the size of the database.

3.1 SCOPE Optimizer

This section talks about the SCOPE's optimization engine in detail. The key feature of the SCOPE optimizer is to incorporate parallelism into the optimization phase. Parallelism is achieved by partitioning the massive dataset into chunks on different machines and perform

computation simultaneously on all machines to reduce the execution time subject to the condition that the parallel computation doesn't affect the correctness of the operation. However, partitioning data and send it to different machines over the shared network is an expensive operation. Thus reducing the number of partitioning operators with not too much execution time is an important optimization goal. In traditional techniques, an efficient serial plan is generated during optimization phase and the parallelism is considered after the optimization step. This may sometimes lead to sub-optimal plan.

In Figure 3.3, a query is given and corresponding to it two distributed plans were generated. Plan (a) first repartition relation R and S on $(R.a, R.b)$ and $(S.a, S.b)$ respectively. Then after doing Hash-Join on $R.a = S.a$ and $R.b = S.b$, optimizer again repartition it into $(R.c, S.d)$ to perform group aggregate. These partitions have been done to parallelize computation as much as possible and reduce the execution skew on each machine. However, in Plan (b) partitioning have been done on $R.a$ and $S.a$ for Hash-Join which is correct since they are the subset of Join columns. If $R.a \rightarrow R.c$ functional dependency exists (which implies that if data is partitioned on $R.a$ then it is also partitioned on $R.c$) then, there is no need of repartitioning and group aggregation can be performed. Now depending on the data size and the selectivity of the query, Plan (b) can be much cheaper. Thus, incorporating parallelism into the optimization phase can improve the execution plan selection.

Now, after setting the motivation for incorporating partitioning into the query optimizer the paper formalizes different structural properties of relation like grouping, sort on some set of columns and the effect of different partitioning schemes (like {Hash, Range, Non-deterministic} Partitioning) on them. For instance if a relation is sorted on a subset of columns C then after Range partitioning on a set of columns C' and $C \Rightarrow C'$ the partitions will still be sorted while if we do Hash partitioning then such properties will not hold. The details of such formalism are omitted in this report [4].

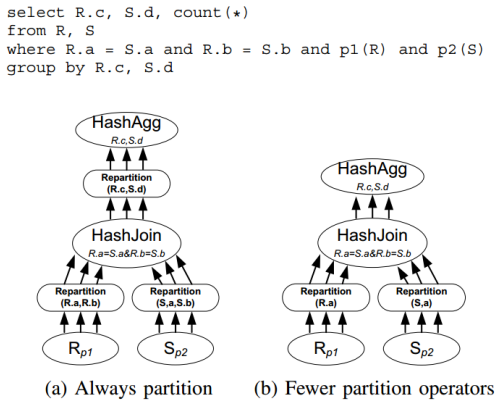


Figure 3.3: Two distributed query plans for the given query [4]

3.2 SCOPE Optimizer Performance

Now to analyze the performance gain from the proposed optimizer I have referred a query from the paper [4]. The script below computes how much machine time has been been spent on jobs issued by different user groups during the last month.

→ extractStart = EXTRACT CurrentTimeStamp, ProcessGUID FROM "[...]/ProcessStartedEvents?Date=(Today-30)..Today" USING EventExtractor("ProcessStarted");

→ startData = SELECT DISTINCT CurrentTimeStamp AS StartTime, ProcessGUID FROM extractStart;

→ extractEnd = EXTRACT CurrentTimeStamp, UserGroupName, ProcessGUID FROM "[...]/ProcessEndedEvents?Date=(Today-30)..Today" USING EventExtractor("ProcessEnded");

→ endData = SELECT DISTINCT CurrentTimeStamp AS EndTime, UserGroupName, ProcessGUID FROM extractEnd;

→ perUserGroup = SELECT SUM((EndTime-StartTime).TotalMilliseconds)/3600000 AS TotalCPUHours,
UserGroupName FROM startData JOIN endData ON startData.ProcessGUID == endData.ProcessGUID GROUP BY UserGroupName ORDER BY UserGroupName;
→ OUTPUT perUserGroup TO "/my/CPUHoursPerUserGroup.txt";

The script [4] first selects events from the last month and extracts time stamp information when each vertex (process) starts and ends, respectively, plus process guid and user group information. Next, duplicates in the raw events are removed by applying a DISTINCT aggregate on all columns. Finally, the cleaned data are joined on the process guid and the total CPU time per user group is calculated.

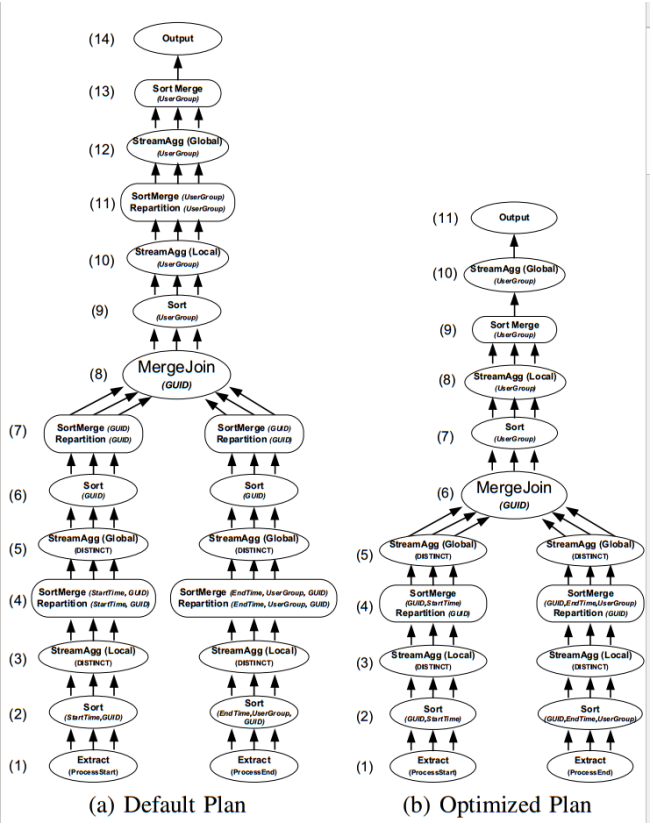


Figure 3.4: Query Plan Comparison [4]

Default Plan: To get DISTINCT StartTime and EndTime, default plan has sorted relation ProcessStart and ProcessEnd on (StartTime, GUID) and (EndTime, UserGroup, GUID) respectively. Then after local DISTINCT operation, relations are partitioned on (StartTime, GUID) and (EndTime, UserGroup, GUID) respectively to perform global DISTINCT operation. After that to perform JOIN operation relations are repartitioned on GUID and the JOIN is performed. Then, local aggregate is performed on UserGroup and then finally relations are repartitioned on UserGroup to perform global aggregate and then merged to the single output.

In this plan 150 partitions are used. It has been done to improve load balancing and avoid data/execution skew on any node.

Optimized Plan: In this plan, initially relation ProcessStart and ProcessEnd are sorted on (GUID, StartTime) and (GUID, EndTime, UserGroup) respectively and local DISTINCT is performed. Then the relations repartitioned on GUID, global DISTINCT operation performed and finally relations are JOINed on GUID. Then instead of repartition on UserGroup, relations are sorted on UserGroup and merged into a single relation and Aggregate is performed.

The optimizer takes advantage of the fact that the relations are fairly small and decided to not repartition on UserGroup. Compared with the default plan, the optimized query has a speedup close to 2x and the optimization cost is negligible compared to the gain.

Authors conclude the paper by mentioning the expressive power of SCOPE in comparison with Map-Reduce model. Its simple programming model by decoupling the optimization from

the user side and incorporating it implicitly into the optimizer expands the user space of the language.

Chapter 4

Query Optimization for Massively Distributed Query Processing

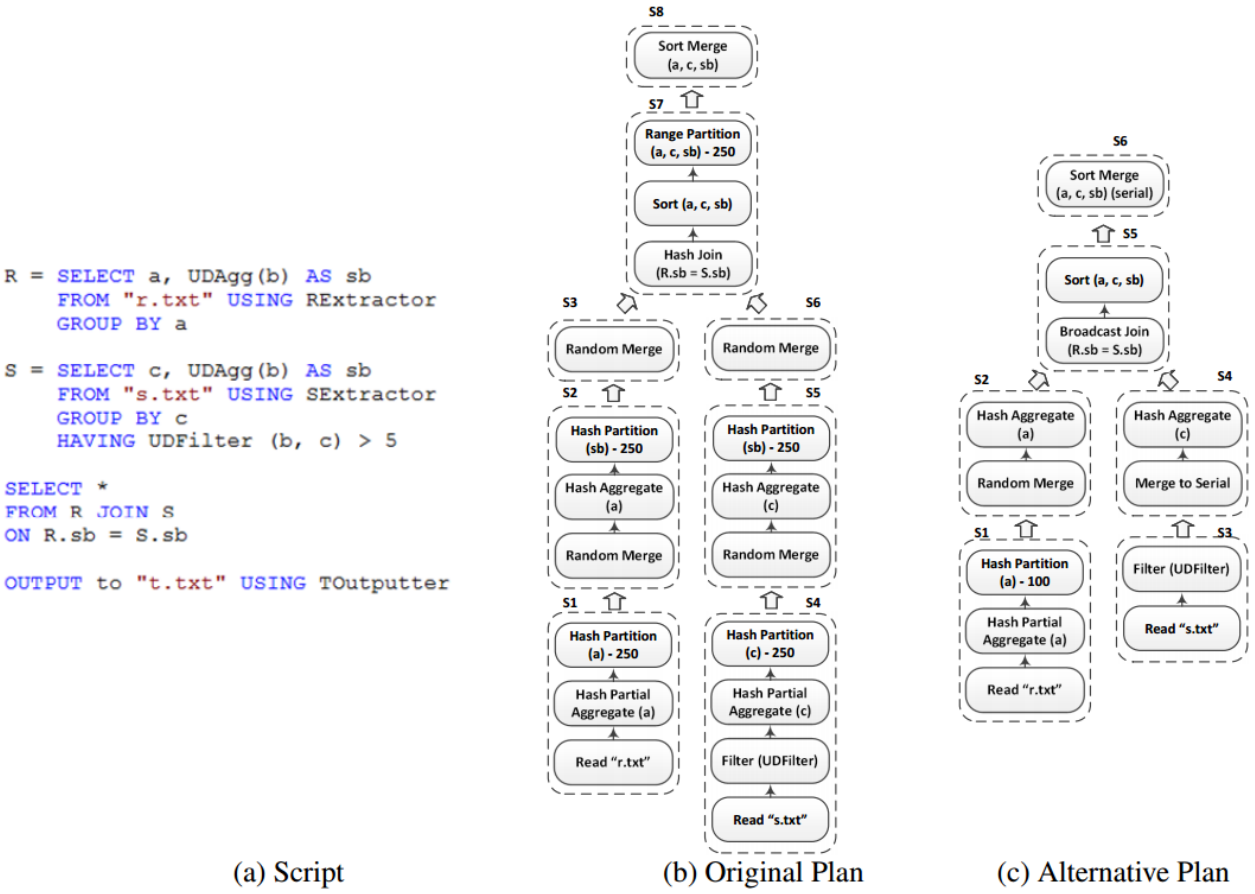
This chapter talks about query optimization techniques for Massively Distributed Query Processing. It covers two optimization techniques: Recurring Query Optimization [5] and Continuous Query Optimization [6].

4.1 Recurring Query Optimization

In the recent years, many applications have an enormous increase demand of compute and storage which leads to distributed data storage and distributed parallel computing. Many programming models have been developed adhere to this demand including Map Reduce model. Though these models highly scales the computation but poses new challenges to query optimization. This optimization technique talks about gathering the statistics of the running query and store it to optimize the re-run of the same query (or query fragment) [5]. One of the key features for this technique is the statistics about the selectivity of a relation, relation size, etc. Accurate collection of these statistics heavily improves the cost of the generated plan by the optimizer.

The importance of these statistics can be understood by the following example from the paper [5].

In Figure 4.1, Figure(a) shows the original plan without the knowledge of any statistics. Firstly, the partition has been done on attribute a and c of the relations generated from the file $r.txt$ and $s.txt$ respectively to perform group aggregate. Then, relations are repartitioned on sb and HashJoin is performed. Finally, relations are sorted, range partitioned and merged into a single relation. These partitions are done to maintain load balancing and reduce data skew. However, if we know the data size then we can reduce the parallelism from 250 to (say) 100 and reduce the resource consumption. In addition to that if the selectivity of UDFilter() is low then we can merge relation S into single relation and do broadcast Join with R and again merge the relation. Therefore, the plan in Figure 5.1(c) is better if the assumed statistics are accurate otherwise it will perform badly. So, the need of accurate statistics is primary for query optimization.



(a) Script (b) Original Plan (c) Alternative Plan

Figure 4.1: Different execution plans for the given query [5]

4.1.1 Optimizer Engine

After compilation a logical query tree is constructed. Then a plan signature is computed over the tree. The Plan signature can uniquely identify a logical query fragment during optimization. During optimization, after each transformation of a logical subtree node the optimizer stores the mapping of resulting transformation with its source and rule. Now using this mapping after optimization, for each logical subtree node we can reach the initial semantically equivalent expression which can be used to represent the operator node. Now, for each subtree, serialize all the expressions below and compute a 64-bit hash which will serve as a plan signature. All the nodes having the same signature will have same properties/statistics. The optimizer stores the mapping of these plan signatures with their statistics in statistics repository.

In the case of parametric queries, if one directly compute their plan signature it will always be different as the change in parameter value will change the plan signature. Therefore to incorporate parametric queries into plan signature, the optimizer change the parameter value with a constant value and compute plan signature on that.

Figure 4.2 represents the different stages of the execution cycle with self-explaining stage title. In

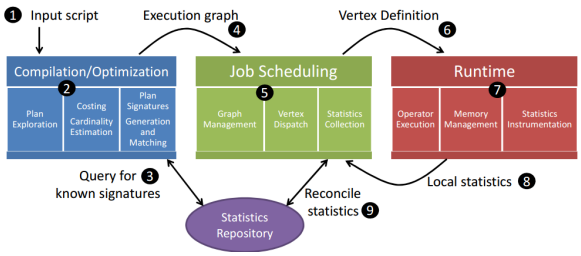


Figure 4.2: Execution environment with statistics collection [5]

stage 3 during optimization phase, optimizer query the statistics repository to find the entry corresponding to a plan signature. If it is present then it is used to generate optimal plan otherwise optimizer generates statistics using heuristics. In further stages the optimizer sends the DAG graph to the Job Manager which schedules and manages the execution of each node of DAG and handles parallelism and failure recovery. During execution the node collects the new statistics and sends the data back to JM which in turn updates the entry in the statistics repository.

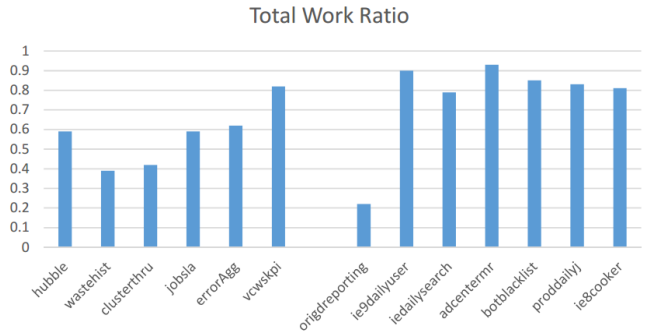


Figure 4.3: Performance evaluation [5]

Performance Analysis: The authors of the paper [5] shows the work ratio graph depicted in Figure 4.3 which represents the fraction of the execution time reduced by introducing recurring query optimization in the execution environment. The machine-hours consumed by the queries is reduced by 10% to 80%.

4.2 Continuous Query Optimization

In the previous section, we describe the work of the paper [5] which talks about a mechanism to store the mapping of plan signature and the corresponding statistics so that optimizer can get the accurate statistics for the recurring queries. However in the cloud-scale environment a single query make take from several minutes to few hours depending on the workload, thus there is a scope to update the query plan as the query is executing. This paper [6] talks about the continuous query optimization where the optimizer continuously monitors the query execution, collect accurate heuristics and adapt the query plan to those statistics.

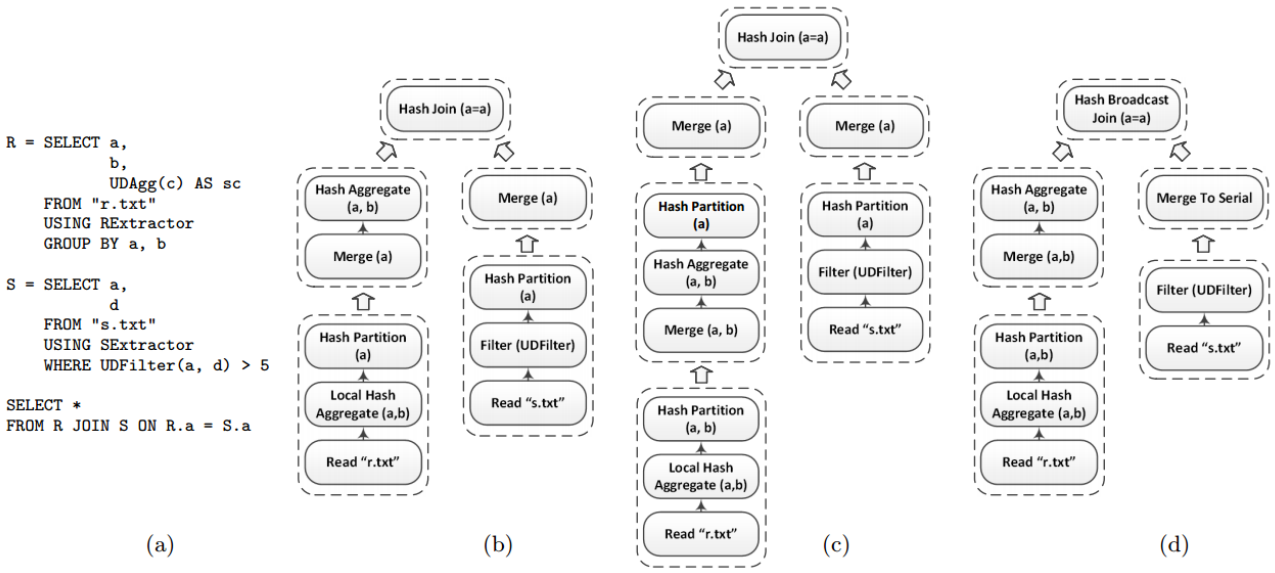


Figure 4.4: Different Execution Plans for a query [6]

Consider a sample query from the paper [6] in Figure 4.4(a) where relation R is formed by extracting "r.txt" and aggregating on columns $\{a, b\}$ and relation S is formed by extracting "s.txt" and filtering on a UDF. After that relation R and S are joined on column $\{a\}$. Now, one of the possible execution plan for the query is given in Figure 4.4(b) where it first reads "r.txt" for relation R, do local hash aggregate on $\{a, b\}$, and then do hash partition on $\{a\}$ so that it can do global aggregate on $\{a, b\}$ and then join on $\{a\}$. Similar policy has been adopted for relation S. This plan is attractive since it refrains relation R to be partitioned twice. However, if there is a data skew on $\{a\}$ then it would be better to first partition on $\{a, b\}$, do global aggregate and then re-partition on $\{a\}$. This plan is demonstrated in Figure 4.4(c). Now, in addition to this, if the optimizer knows that the selectivity of UDF filter is very low then instead of partitioning S on $\{a\}$, it can merge it into a single relation and do broadcast join with relation R. This will in turn remove the necessity of re-partitioning R on $\{a\}$ but if the above statistics are not accurate then this will perform terribly. Hence, the accurate statistics are primary to query optimization.

Since the queries in the cloud-scale distributed environment takes the time in the order of minutes or hours, the overhead of query optimization is very low as compared to the traditional single machine systems where queries runs typically in the order of milliseconds. Hence, it opens the large room for query optimization. Another advantage is that due to distributed nature of the execution, it divides whole execution into stages, materialize intermediate results between stages to reduce fault recovery time which also opens optimization opportunities as execution of a stage begins when all its inputs are available and optimizer can adapt its execution plan based on the statistics from the previously executed stages while traditional systems uses pipeline to its extreme and there is no need to divide into stages. However within a stage, the execution is in pipeline fashion. The optimizer treats a stage as a unit of computation.

4.2.1 Optimizer Architecture

Figure 4.5 highlights the different stages of the execution cycle. It is similar to the architecture from the previous paper [5] except stages 7, 8 and 9. In stage 7, instead of storing the statistics in the statistics repository it transfers the collected statistics to the optimizer. In stage 8, optimizer adapts the current plan with the freshly obtained statistics and transfers the new plan to the JM which then merges it with the current plan and adapts the execution of newly formed nodes.

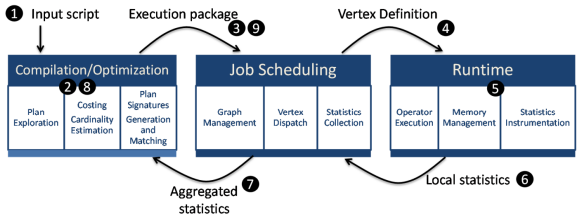


Figure 4.5: Continuous Optimization Architecture [6]

4.2.2 Optimization

Plan Signature

Plan Signatures are computed exactly as mentioned in the previous section. Figure 4.6 shows plan signatures of different nodes of a query tree as a 64-bit hash value. However, since the partition and merge operator will have the same signature as they have same logical representation with different properties. So, to differentiate between them the optimizer appends delivered properties with the signature and then compute the hash value.

Statistics

As mentioned before, statistics are collected for re-optimization. However, there has to be certain characteristics of the statistics. First, collection and storage of the statistics should have low overhead. Second, statistics must be composable as each vertex collect partial statistics and send them to JM which aggregates them. Different statistics were collected including:

- **Cardinality and average row size:** Cardinality is calculated by counting total number of rows and average row size is equal to the size of relation divided by the total number of rows.
- **User Defined Operator cost:** A timer has been set when a UDF is executed and noted after its completion. total elapsed time is calculated to find the cost.
- **Partitioning Histogram:** For the partitioning operator, total number of rows in each partition are computed for the histogram.

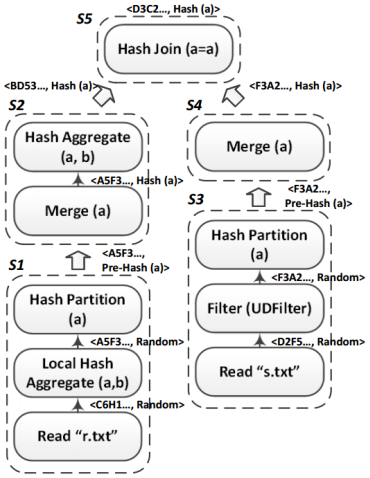


Figure 4.6: Plan Signatures of nodes [6]

Data Structures

There are two data structures used in the optimization engine, *Statistics package* and *Materialization package*. Statistics package contains the mapping of all plan signatures with their corresponding collected statistics returned by the executed vertices.

Materialization package contains the information of the intermediate materialized views.

Statistics package is updated for every plan signature in a completed vertex. Figure 4.7 from the paper [6] shows a pseudocode executed after each vertex completion. Due to failures a vertex might be executed multiple times hence it only update statistics package and materialization package when the vertex got completed for the first time. Then if the re-optimization policy suggests to update the current plan, a new plan is created based on the SP and MP and if it differs from the current plan then it is sent to the JM.

```
ON VertexCompleted (V: vertex in stage S)
01  SP = global statistics package
02  MP = global materialization package
03  if (V is completed for the first time)
04     updateStatisticsPack(SP, V.stats)
05     updateMaterializationPack(MP, V)
06  if (reoptimization policy)
07     newPlan = optimize(query, SP, MP)
08     if (newPlan != currentPlan)
09        replace currentPlan with newPlan
10   // additional vertexCompleted handling...
```

Figure 4.7: Pseudocode executed after vertex completion [6]

Partial Materialized Views: In Materialization package, besides the information of materialized views of the completed stages, optimizer also keeps the materialized views information of the executing stages. In VertexCompleted method, during generation of a new plan, it considers the materialization information of the vertices that are currently executing since, if the cost of the new plan is more than the cost of the current plan minus the cost of the work that has already been done then it sticks with the current plan.

4.2.3 Performance Evaluation

Figure 4.8 from the paper [6] represents a sample query used to judge the performance of the proposed optimizations. Figure 4.9 represents the different execution plan generated due to multiple plan adaptation. Figure 4.9(a) shows the initial plan generated without any knowledge of the data properties. When Stage 0 : $S1$ got executed, it gives the statistics that the selectivity of UDF filter is very low then the plan is changed with the reduced degree of parallelism for Stages $S2$ shown in Figure 4.9(b). After Rack merge operation in $S1$ it gives the statistics that there is a data skew on column a. Hence a new plan is generated with partition on columns a, b shown in Figure 4.9(c).

```
A = SELECT a, b, UDAgg(c) AS sc
    FROM "r.txt" USING UDExtractor()
    WHERE UDFilter(a, b) > 10
    GROUP BY a, b

SELECT a, UDAgg(b) AS sb, UDAgg(sc) AS ssc
FROM A
GROUP BY a
```

Figure 4.8: Sample Query [6]

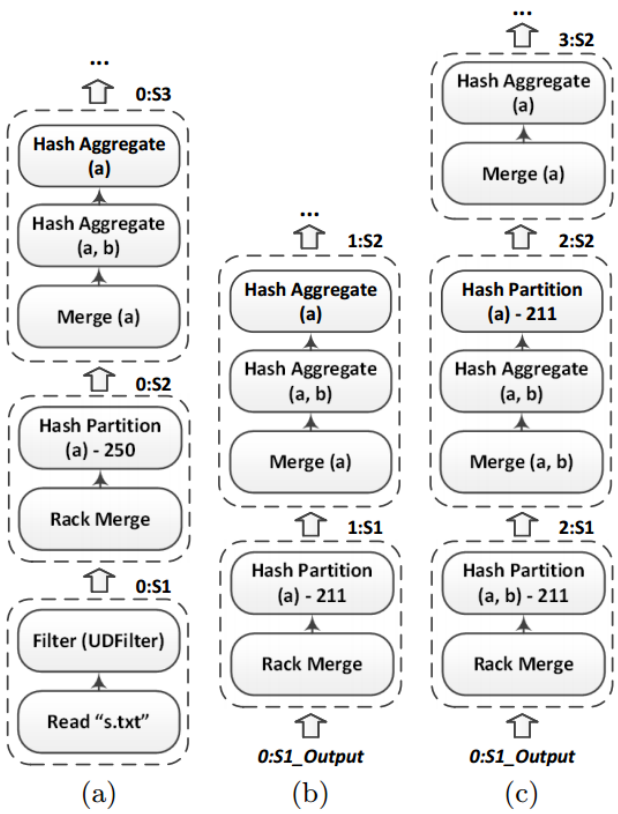


Figure 4.9: Different execution plan for the given query [6]

Figure 4.10 represents the normalized execution latency with Baseline as the latency of the original optimizer. Second and third value shows the normalized latency using CQO(Continuous Query Optimization). The results shows the query latency is reduced by 8.4x.

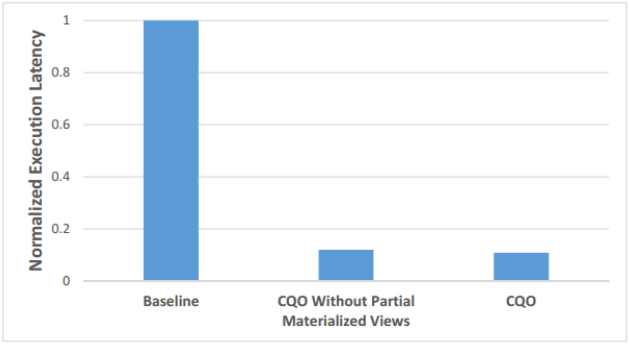


Figure 4.10: Query Latency [6]

Chapter 5

Query Optimization for Streaming Data

Till now, we have seen the different optimization techniques for massive distributive data. However, all these optimizations were based on batch(static) data. Now, this chapter will explore the work of many papers on the query optimizations on streaming data. Streaming data is different from batch data as it is unbounded stream of data and the timely nature is an important aspect of this data.

5.1 Apache Flink^{TM}: Stream and Batch Processing in a Single Engine

Data-stream processing and batch processing are considered two different programming paradigms which are executed in different environment, by different systems and using different API(s). The dedicated streaming systems are Apache Storm, Google Cloud Dataflow, Microsoft StreamSight and IBM Infosphere streams while dedicated batch processing systems are Apache Hadoop, Apache Spark etc. Most of the business usecases used to be based on batch processing systems. in recent years, there has been a huge increase in the demand for the stream processing systems for the analysis of click streams, sensor data, web/application logs. Many systems batch these stream data into finite size chunks and process them. They ignore the timely nature of the stream data.

Apache Flink [7] introduces a paradigm of a unified model which can execute stream data processing in real time(by keeping its timely nature) and batch processing in the same programming model and execution engine. It creates API(s) abstraction over its execution engine to provide such functionality. Flink treats batch data as special case of stream data when streams are finite and the order and time of the data doesn't matter. However, there are additional optimizations for batch processing and specific scheduling mechanisms are employed for efficient execution.

5.1.1 Flink Architecture

Figure 5.1 shows the different stack layers of the Apache Flink Architecture. Core layer contains the runtime environment common for stream data processing and batch data processing. API & Libraries layer shows two types of API(s) for both paradigms. On top of this bundles bunch

of applications in Machine Learning, Graph processing, Table processing(using DataSet API for batch processing) and similarly for stream processin

Figure 5.2 from the paper [7] shows that the Flink runtime environment comprises of a client, a Job Manager and at least one Task Manager. The client takes the program code, generate a Dataflow graph(DAG) and send it to the Job Manager. For the DataSet programs(batch programs) additional cost-based optimization is done by the client. The Job Manager coordinates the distributed execution of the dataflow. It schedules the nodes(tasks) of the Dataflow Graph on any Task manager, monitors their status, and recover the failure of the nodes. It also manages checkpointing to ensure bounded recovery time.

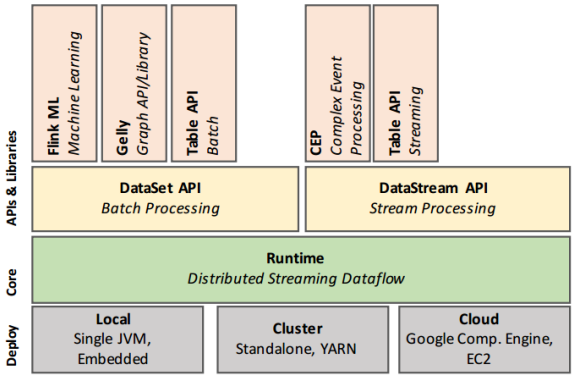


Figure 5.1: Flink Architecture [7]

Data Exchange among nodes: Data can be exchanged between two operators in pipelined manner or blocking manner. In pipelined data exchange, the producer continuously sends data to the consumer with the consumer applying the back pressure with some buffer (to compensate short-term throughput drops). However, some operator cannot operate in pipelined manner like sort-merge operator so we can use blocking data exchange when the operator transmits data only when it is completely ready. This choice depends on the operator and transparent to the engine.

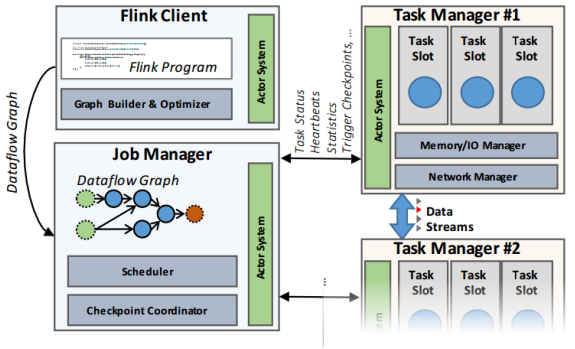


Figure 5.2: Flink Runtime Environment [7]

The producer maintains a buffer and sends data to consumer after buffer is full or after the timeout, whichever is earlier. This way it can manage *high throughput* by choosing buffer size large enough and *low latency* by using appropriate timeout value.

5.1.2 Fault Tolerance in Flink

Apache Flink offers a strict exactly-once-processing consistency guarantees. The assumption taken by Flink to achieve this is that the data source is persistent and re-playable. However, it can also deal with non-persistent sources by using write ahead log of the data generated at source. Flink uses Asynchronous Barrier Snapshotting(ABS) by using control events. Barriers are the control records inserted into the data streams at fixed interval of time. Barriers logically divide streams into finite sized chunks. When a operator receives a barrier from a input stream it waits for the barrier events from all the inputs, it then persist its state on a durable storage and then forwards the barrier downstream. Through this mechanism every operator will eventually persists its state for a given barrier. Now, during failure Flink can recover from the last snapshot at each operator. Some data processing might have to be done again but it is bounded by the data size between two barrier events.

This method also enables to implement the *versioning mechanism* by using persist states as save-points and reload the historic states of the data when needed.

5.1.3 Memory Management

Flink implements its own memory management by serializing data into memory segments instead of using JVM heap objects. This results in high memory utilization and low garbage collection overhead. However, this incurs an extra complexity of serialization and de-serialization of data which is kept at minimum.

5.1.4 Conclusion

In the paper [7], Authors have presented Apache Flink's as a unified model for stream and batch precessing. It has a common runtime engine which is utilized by both processing paradigms in different ways. The DataStream API provides the time bound recovery using ABS mechanism, keeps the state persistent and provide windowing mechanisms. While the DataSet API provides an efficient execution using query optimizer and blocking data exchange by spilling data on disk during memory overflow.

5.2 On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines

Stream Processing systems must be resilient to the load changes in the unbounded incoming stream data. However, most of these systems (like Apache Flink, Apache Storm) provides little to no functionality regarding the dynamic adaptation of the system. One needs to restart the query/processing from the previous checkpoint to change any system configurations including adding new operators or changing their parallelism. In this paper [9] authors proposed a mechanism to incorporate these changes into the system without ever stopping the query. The authors mainly proposes three modification protocols for i) migrating an operator instance(s) in case of the node failure, ii) adding a new operator instance into the system, and iii) changing the UDF on an operator instance.

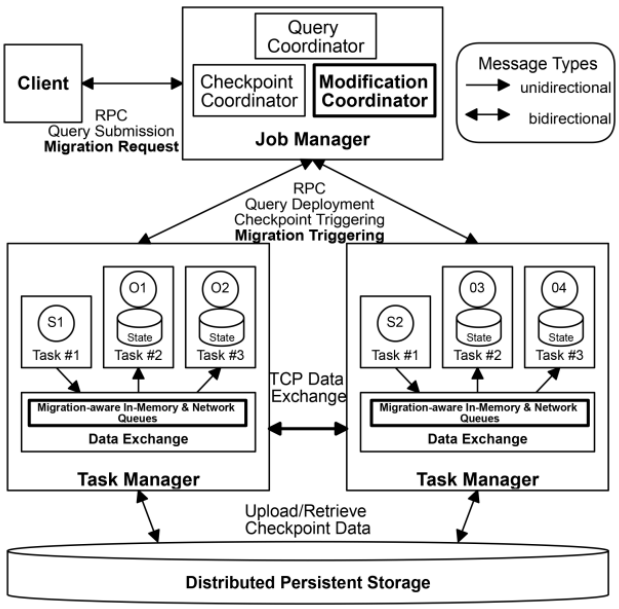


Figure 5.3: Flink Updated Architecture [9]

5.2.1 Protocol in Detail

To incorporate the above protocols in Apache Flink, Flink's architecture has been changed by introducing Modification Coordinator in Job

Manager which initiates the modification protocol and adding some extra code in Task Manager to make its resources migration-aware. Figure 5.3 from the paper shows the above mentioned changes into the system.

Migration Protocol

The client submits a migration query to the Modification Coordinator. It then injects a modification marker (which contains all the instructions regarding the migration) into the data stream at the data source. The marker eventually reaches to every operator instance. Each operator instance independently react to the marker. The upstream operators (operators which directly send their output stream to the migrating operator) wait for the upcoming checkpoint barrier to trigger the migration process. After checkpoint barrier they stop sending the records to the migrating operator and starts buffering the records in the migration buffer and gracefully spills to the disk in case of overflow. After this, migrating operator safely instantiated on different node. For the downstream operator (operators which directly receives their input stream from the migrating operator), migrating operator sends a migration message in the end before migrating and based on the message downstream operators adjust their connection with the new migrated node. Figure 5.4 from the paper [9] shows the working of the protocol.

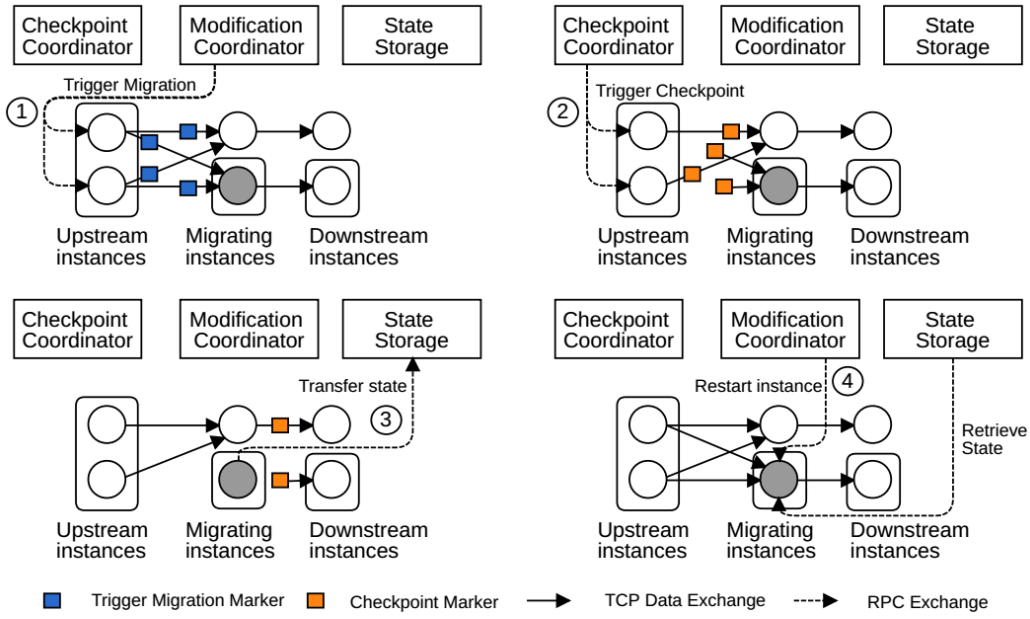


Figure 5.4: Modification Protocol Overview [9]

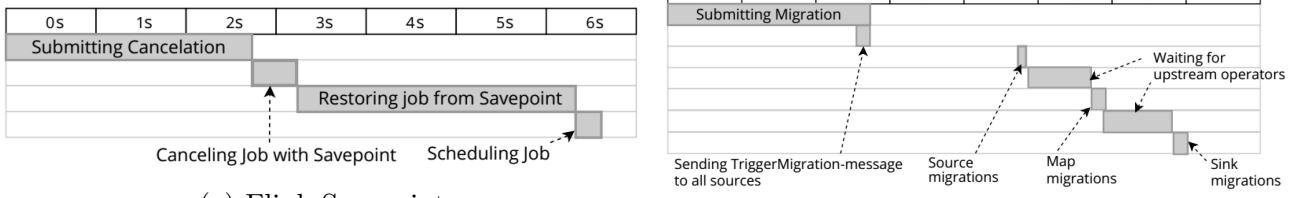
New Operator Instantiation

Introduction of new operator works similarly like the migration protocol. In this case a new operator is being introduced between two existing operators. After the operator(s) which acts as an upstream operator stops sending their records, the new operator is instantiated and similarly downstream operator(s) are notified by the upstream operator(s) in their last message and hence downstream operator(s) adjust their connection with the new operator node(s).

Changing the operator's UDF

Similar to the new operator instantiation, the user provides the code for the new UDF and each task manager involved in the updation process fetches the code. Afterwards each operator instantiate the new UDF and registers a callback. The operator resumes operation after each operator resister the callback to ensure consistency.

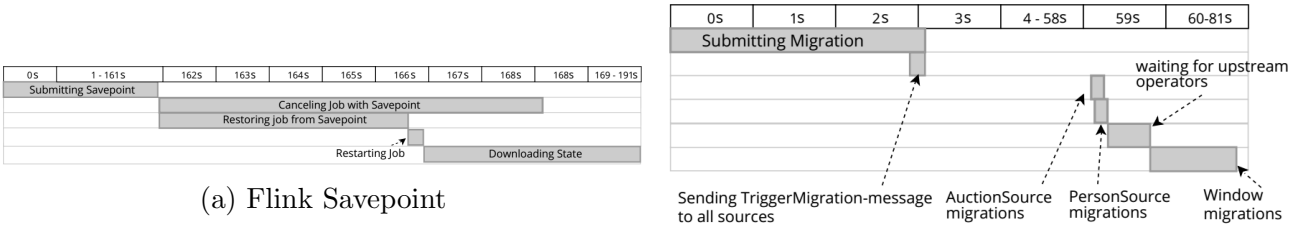
5.2.2 Performance Study



(a) Flink Savepoint

(b) Migration Protocol

Figure 5.5: Total time taken for Query 1 [9]



(a) Flink Savepoint

(b) Migration Protocol

Figure 5.6: Total time taken for Query 2 [9]

The experiment was done on two queries taken from the paper [9] where the total state size of the operators of the Query 1 was 100 KB and the total state size of the operators of the Query 2 was 13.4 GB. Figure 5.5 and Figure 5.6 from the paper shows the comparison of the total time taken in the process for the Query 1 and Query 2 with the savepoint mechanism and migration protocol. For Query 1, Migration protocol took comparable time in comparison of savepoint mechanism while in Query 2, Migration protocol significantly perform better.

The authors concluded that Modification protocol is beneficial over savepoint mechanism in the case of large operator state. Modification protocol also opens the scope for continuous optimizations for the steam processing engines and adjusting the systems' resources based on the workload.

5.3 On Fault Tolerance for Distributed Iterative Dataflow Processing

Large scale graph processing systems and machine learning based analytics uses distributed iterative processing. Multiple iterations needs to be done for the machine learning algorithms

to converge or touch a threshold. Failures are inevitable in such systems and one can't use optimistic recovery by restarting the whole computation after the failure as these systems runs for a long time and have a very large state. So checkpointing mechanism are used to periodically persist the state of each node in the system on a durable storage like HDFS. During checkpointing the operator stops its output stream to the downstream operator and save its state. This incurs high latency due to blocking checkpointing mechanism. During failure recovery, all the operator instances restores their states to the last checkpoint and restart their computation. This is referred as complete recovery and has high overhead in case of the failure of very few nodes. For all these reasons authors of the paper [10] proposes an unblocking checkpointing mechanism and confined recovery mechanism.

5.3.1 Unblocking Checkpointing Mechanism

The unblocking checkpointing is an iteration aware checkpointing mechanism where the checkpointing is done between every two iterations. The authors proposed two unblocking checkpointing mechanisms: Tail checkpointing and Head checkpointing. Figure 5.7 depicts the same.

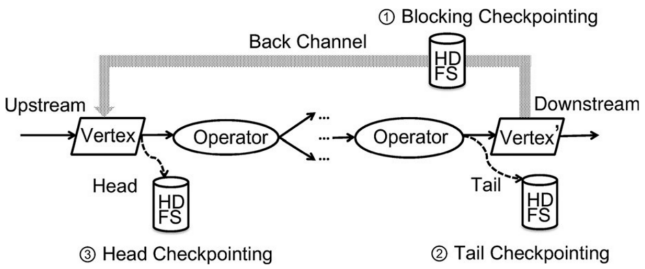


Figure 5.7: Different Checkpointing Mechanisms [10]

Tail Checkpointing

In Tail Checkpointing, the iteration coordinator continuously checkpoints the data parallely as it writes the data for the new vertex V' (i.e, vertex for the next iteration). If the checkpoint speed is more than the streaming rate to produce V' then all the data of the i^{th} iteration will be check-pointed when the new vertex is ready for the $(i+1)^{th}$ iteration and there will be no checkpointing overhead at all. If streaming rate is higher than there will be some checkpointing overhead which will be less than the overhead in blocking checkpointing.

Head Checkpointing

In Head checkpointing, the iteration coordinator checkpoints the $(i-1)^{th}$ iteration state during i^{th} iteration. Similar to the Tail checkpointing, if the checkpoint speed is more than the streaming rate to produce V' then there will be no checkpointing overhead. However, there is still no runtime overhead if the difference between the checkpointing time and streaming time of vertex V' is less than the processing time for the i^{th} iteration. Therefore, the overhead incurred by Head checkpointing will be less than that of Tail checkpointing. However, in Head Checkpointing after the i^{th} iteration coordinator has checkpointed till $(i-1)^{th}$ iteration while in Tail Checkpointing it has checkpointed till i^{th} iteration. Hence, during recovery Head checkpointing might incur more overhead.

The details of these mechanisms and their latency analysis is explained in the paper [10].

5.3.2 Confined Recovery

In the case of complete recovery all nodes restores their state to the last checkpoint state and starts their re-computation. It seems quite wasteful in case of a single or few nodes failure. Authors introduced a confined recovery mechanism which requires only failed nodes to restore from the last checkpoint and recompute till the current iteration. To do this all the states log the output they send to their neighboring nodes. Now, during recovery the failed nodes retrieve their inputs from the neighboring nodes to recompute its lost state. Details of the mechanism is omitted from this report.

5.3.3 Performance Evaluation

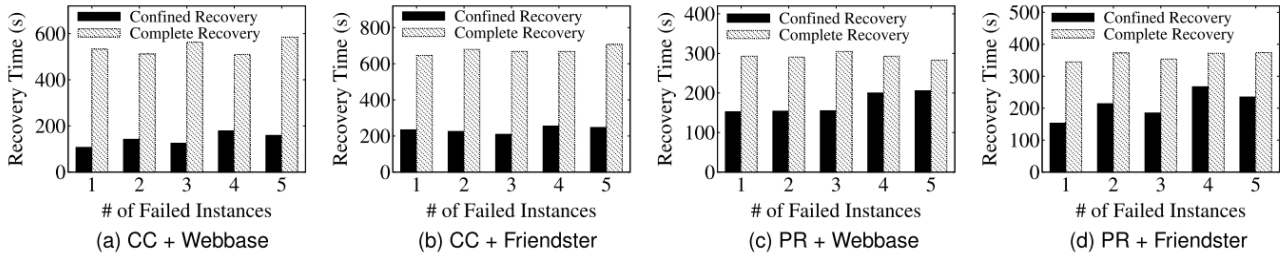


Figure 5.8: Checkpoint Performance Comparison [10]

Figure 5.9 shows the execution time of different checkpointing mechanisms over two algorithms: Connected Components(CC) algorithm and PageRank algorithm. It uses two datasets for both the algorithms. The authors conclude that the unblocking checkpointing mechanism outperforms blocking checkpointing mechanism and in that Head checkpointing works slightly better. Figure 5.8 shows the comparison between confined recovery time and complete recovery time on these algorithms and datasets which clearly shows that confined recovery works significantly better than the complete recovery.

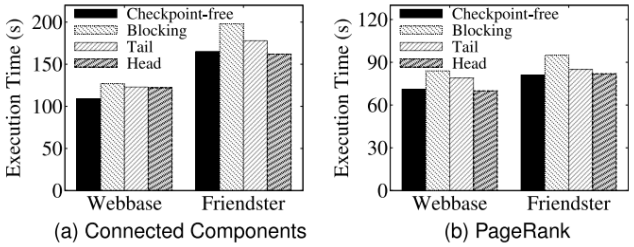


Figure 5.9: Checkpoint Performance Comparison [10]

5.4 The NebulaStream Platform for Data and Application Management in the Internet of Things

Over the recent years, the amount of data gathered has increased by many folds as compared to the past. Most of these data comprises of sensor data, mobile data and the data from millions of IoT devices. The processing of this data has posed many challenges to the existing processing systems for big-data like Apache Flink, Hadoop etc. Hence, there is a need to develop novel processing model which have low latency, location awareness because of the huge geographical diversity of the IoT devices and real-time processing on these devices.

However, current state of the art systems are either cloud-based or fog-based to process data from the IoT devices. In cloud based systems, the data gathered from all the devices collected at a central unit where its processing is done. These systems don't exploit the capabilities of IoT devices. These systems do not scale with the number of IoT devices and hence can't hold their ground in the future. To illustrate this, the authors of the paper [8] presented a curve between latency and number of IoT devices in Figure 5.10 executed on a Flink cluster. The curve depicts that upto 10 producers the curve flattens with time but for more than 10 produces the latency continuously increases with time. This experiment shows that cloud centric systems do not scale with the number of IoT devices.

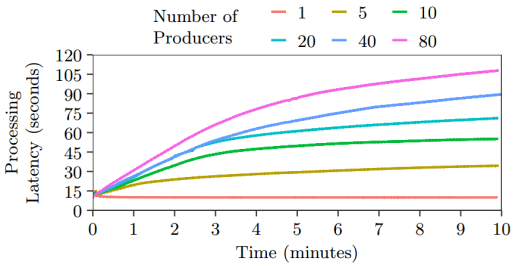


Figure 5.10: Latency vs Time in Flink [8]

Systems based on fog environment exploit the processing capabilities of Edge devices connected with the sensors/source. These devices do early filtering and aggregation whenever possible to reduce the data volume in the execution pipeline. However, these systems scales within the fog and do not exploit the virtually unlimited resources of the cloud infrastructure. Hence, there does not exists a unified system which combines sensors, fog and the cloud environment. The authors proposes NebulaStream Platform [8] to resolve the same. However, to develop such system there are three challenges mentioned below which needs to be addressed.

1. **Heterogeneity:** A unified system will be highly diverse based on the storage and processing capabilities ranging from sensors/edge devices with no storage and negligible compute capability to large cloud clusters with virtually unlimited storage and compute. Current data processing systems don't focus on the hardware tailored code and abstract underlying hardware from the development process. However, efficient utilization of hardware is necessary to provide low latency.

2. **Unreliability:** Due to the highly dynamic nature of the system with unreliable nodes continuous changing their movement (like car sensors) resulting in transient failures effecting the throughput and latency of small devices.

3. **Elasticity:** Since the solution has to integrate multiple environments, it has to deal with different topologies of these models. In fog based environment, data moves from source to neighboring nodes and eventually reaches the root node. Thus data is not available to all nodes all the time while in cloud systems data is centrally available to all nodes though a distributed file system like HDFS.

5.4.1 Nebula Stream Architecture

Nebula Stream Topology shown in the Figure 5.11 have three layers. First layer is the sensor layer where millions of sensors sends data to their immediate connect node (often called as Entry Node). To reduce data volume and save sensor's battery, NES schedules sensor reads based on the user's query instead of continuously reading data. From the sensor layer data enters to the second layer called the fog layer where data is routed from the entry nodes of the fog layer towards the exit nodes with processing data as much as possible. The processing capability of

each node increases as it moves towards the third layer, the cloud layer. When the data reaches the cloud layer, remaining processing is done and the results are outputted to the user.

NES coordinator is responsible for sending and receiving queries to the NES topology. Query is submitted through an API, NES Query Manager generates a logical plan of the query, NES optimizer generates an execution plan based on the current NES topology and gives it to the Deployment Manager. Deployment Manager deploys each fragment of the query on individual nodes. To get the current topology of the system, a feedback loop is connected with NES Topology manager which continuously updates the NES optimizer about the current topology. While the NES topology is decentralized and distributed, NES coordinator is a central entity. Future work of this paper [8] includes the possible decentralization of NES coordinator and making each fog or cloud node potentially a source/sink.

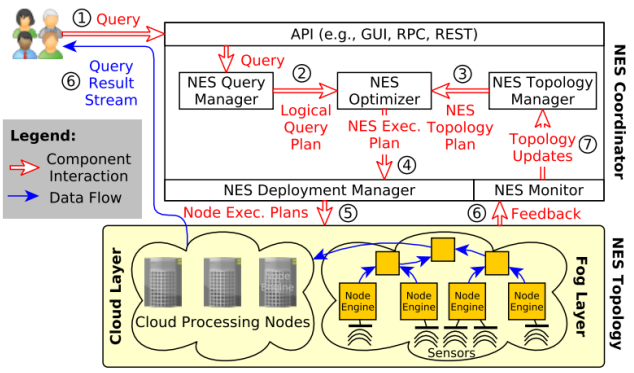


Figure 5.11: NES Architecture [8]

5.4.2 NES Solutions for IoT Challenges

Heterogeneity

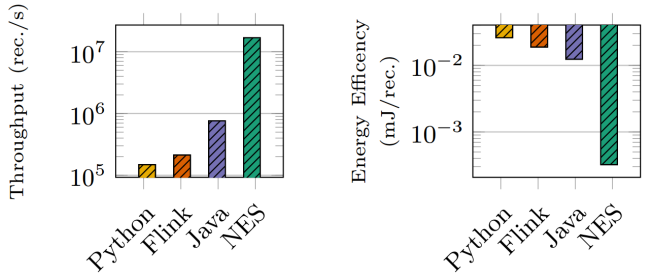


Figure 5.12: Throughput and Energy improvement in NES [8]

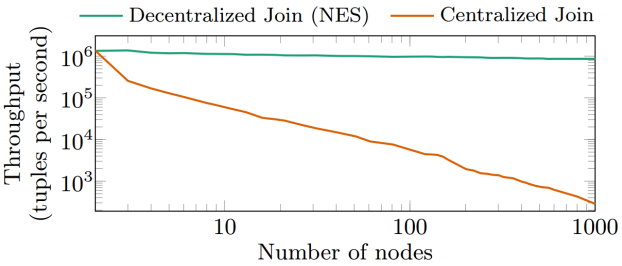


Figure 5.13: Logical Join from NES nodes [8]

To address heterogeneity in data storage and compute, NES applies maximum sharing at three levels. First at a query level, NES exploits data sharing among multiple queries. At operator level, it slices the input streams and share stream aggregations. Finally at sensor level, it uses on-demand scheduling of sensor reads and transmissions. For instance if a sensor property doesn't satisfy the condition in a where clause then there is no need to read its value. It also adds some read tolerance so that during higher query rates it can send the same value to multiple queries by setting the upper bound on the time duration between them. Figure 5.12 shows the throughput and energy utilized by different frameworks ranging from simple Python system to Flink to NES.

NES exploits hardware resources of million of IoT devices efficiency. It generates hardware-tailored code for each device. This reduces the query optimization overhead as NES optimizer only generates a high level optimized plan and leave the further optimization on each node.

Furthermore, NES utilizes the in-network processing in fog layer to reduce the computation overhead at cloud layer. For instance, Figure 5.13 shows the throughput comparison between centralized join and decentralized join. As the number of nodes increases, in case of centralized join data stream also increases which lead to decrease in the throughput. However in decentralized join, as the node increases so does the total processing power of the system which results in constant throughput. This shows that decentralized computation scales well with the IoT devices.

Unreliability & Elasticity

NebulaStream's goal is to provide consistency and availability despite frequent failure of unreliable nodes. Furthermore, moving sources continuously change their connections which results in transient failures. To deal with this, NES employed fine grained recovery protocol. At sensor layer, it substitutes the missing/broken sensor read from the neighboring sensor. At fog layer, data has been sent through multiple paths to make sure data reach to the destination in presence of some nodes failures. At cloud layer, there already exits fault tolerant mechanisms like global checkpointing. To ensure continuous operation of a node during constant evolution, NES equips a node with alternate plans so that node can choose any plan based on its topology.

5.5 Mosaics in Big Data: Stratosphere, Apache Flink, and Beyond

This paper [11] talks about the challenges faced by the Data Science community due to the ever increasing heterogeneity in database systems. Since there has been a large increase in the volume, velocity and veracity of the data, different specialized analytics systems has been developed to cater a specific requirement efficiently. For instance, relational database systems includes (Postgres, MySQL, MongoDB), numerical analysis systems includes (MATLAB, R), distributed batch processing engines includes (Hadoop, Spark), distributed key-value stores includes (Redis, Cassandra), and distributed graph processing systems includes (Neo4j). Moreover in terms of hardware there has been a great development in diverse architectures including CPUs, GPUs, and FPGAs. There has been a economical high availability of Flash storage and new emergence of non-volatile memory (NVMe) which has disrupted the architecture of memory models.

Due to the above stated reasons, the performance of data analytics systems is improving but so does its complexity. Today's data analysts should be aware of all the data processing models and their corresponding trade-offs in different parameters, and must be able to map their analysis task with the best combination of these systems and most efficient hardware configuration. This has increased the entry barrier in the data science community and restricted this domain to the experts.

5.5.1 Mosaics

Due to above stated reasons, Markl [11] proposes to automate all the systems and hardware tuning required in any data analysis task. To do this, he proposes to develop a principled model to represent these systems like the relational algebra for traditional database systems. He decomposed the overall objective into six sub-goals:

1. **Unifying Modeling Across Theories:** Develop algebras equipped with equation theories to represent collections, graphs and matrices in a single model.

2. **Cross Theory Optimization:** Identify the class of interesting optimizations in different systems and find the relations among them to enable cross theory optimization.

3. **Optimization across Engines:** Define a single execution models which encodes the processing and storage capabilities of different engines, and physical properties of the data inputs assumed by these engines. By doing this, one can generate efficient execution plan spanning through multiple engines.

4. **Predicting and Learning Program Runtime:** To develop the optimization capabilities spanning across theories and engines, one needs to develop a novel cost model which can help to choose the efficient plan by comparing plans of different engines.

5. **Optimizing Across Hardware:** Develop a model for the automatic selection of the best hardware configuration subject to performance and budget for the given task and adjusting the plan dynamically for a specific hardware configuration.

6. **Generating Hardware-Targeted Code:** Many systems today solely focuses on scalability rather than the efficient execution of a code. Develop a model to generate a hardware tailored code to fully utilize the underlying hardware for selected tasks. It can be exploited by observing the code generation by the hardware and alter it according to our needs.

The author presents Emma as a mosaics of theories and systems which converts Scala program into big data analytics program for distributed and parallel execution without developers worrying about its API(s). He also presents couple of other developing systems to meet the objective of Mosaics.

Bibliography

[1] Prasan Roy: PhD Thesis
https://www.cse.iitb.ac.in/infolab/Data/Courses/CS632/2015/Papers/prasan-thesis.pdf

[2] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00). Association for Computing Machinery, New York, NY, USA, 249–260. DOI:https://doi.org/10.1145/342009.335419

[3] Chaiken, Ronnie & Jenkins, Bob & Larson, Per-Åke & Ramsey, Bill & Shakib, Darren & Weaver, Simon & Zhou, Jingren. (2008). SCOPE: Easy and efficient parallel processing of massive data sets. Proceedings of the VLDB Endowment. 1. 1265-1276 10.14778/1454159.1454166.

[4] Zhou, Jingren & Larson, Per-Åke & Chaiken, Ronnie. (2010). Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. 1060 - 1071. 10.1109/ICDE.2010.5447802

[5] Bruno, Nicolas et al. "Recurring Job Optimization for Massively Distributed Query Processing." IEEE Data Eng. Bull. 36 (2013): 46-55

[6] Bruno, Nicolas & Jain, Sapna & Zhou, Jingren. (2013). Continuous cloud-scale query optimization and processing. Proceedings of the VLDB Endowment. 6. 961-972. 10.14778/2536222.2536223

[7] Carbone, Paris & Katsifodimos, Asterios & Kth, † & Sweden, Sics & Ewen, Stephan & Markl, Volker & Haridi, Seif & Tzoumas, Kostas. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin. 38

[8] Zeuch, Steffen et al. "The NebulaStream Platform for Data and Application Management in the Internet of Things." ArXiv abs/1910.07867 (2020): n. pag.

[9] Bartnik, A., Del Monte, B., Rabl, T. & Markl, V., (2019). On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines. In: Grust, T., Naumann, F., Böhm, A., Lehner, W., Härder, T., Rahm, E., Heuer, A., Klettke, M. & Meyer, H. (Hrsg.), BTW 2019. Gesellschaft für Informatik, Bonn. (S. 127-146). DOI: 10.18420/btw2019-09

[10] Xu, Chen & Holzemer, Markus & Kaul, Manohar & Soto, Juan & Markl, Volker. (2017). On Fault Tolerance for Distributed Iterative Dataflow Processing. IEEE Transactions on Knowledge and Data Engineering. PP. 1-1. 10.1109/TKDE.2017.2690431

[11] Volker Markl. 2018. Mosaics in Big Data: Stratosphere, Apache Flink, and Beyond. In Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS '18). Association for Computing Machinery, New York, NY, USA, 7–13. DOI: https://doi.org/10.1145/3210284.3214344

[12] G. Graefe and W. J. McKenna, "The Volcano optimizer generator: extensibility and efficient search," Proceedings of IEEE 9th International Conference on Data Engineering, Vienna, Austria, 1993, pp. 209-218, doi: 10.1109/ICDE.1993.344061

[13] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. ACM Trans. Database Syst. 1, 2 (June 1976), 97–137. DOI:https://doi.org/10.1145/320455.320457