# Cost-based Optimization in Parallel Data Frameworks

**B.Tech. Project 1st Stage Report**

Submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Technology (Honors)**

*Student:*
**Pararth Shah**
**Roll No. 09005009**

*Guide:*
**Dr. S. Sudarshan**


Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

**Abstract**

The advent of Big Data has led to the increasing use of parallel and distributed frameworks for data processing. The increasing need for Online Analytical Processsing (OLAP) queries has made it necessary for database systems to handle complex queries with minimal response times. Although query optimization has consistently shown significant performance gains for traditional database systems, the optimization of queries on parallel databases has a lot of room for improvement. It is evident that improvements in query optimization techniques will lead to substantial gains for parallel data frameworks. This report surveys the state-of-the-art in traditional query optimization, followed by a survey of recent parallel data frameworks. We present methods that have been used successfully in traditional database systems like duplicate-free join enumeration and pruning techniques, and present extensions that should be applicable in a parallel setting.

# Acknowledgment

I would like to extend my heartfelt gratitude to my guide, Dr. S. Sudarshan and acknowledge his constant guidance and support throughout the project. I am extremely grateful to him for spending his valuable time with me for clarifying my doubts whenever I approached him.

Pararth Shah
B.Tech. IV
CSE, IIT Bombay

# Contents


**1 Introduction** **4**
1.1 Outline . . . . . 4

**2 Query Optimization Overview** **6**
2.1 Space of Logically Equivalent Plans . . . . . 6
2.1.1 Logical plan space . . . . . 6
2.1.2 Physical plan space . . . . . 7
2.2 Cost Model . . . . . 7
2.3 Search For Optimal Plan . . . . . 7

**3 Survey Of Traditional Optimizers** **8**
3.1 System R and Starburst . . . . . 8
3.2 Exodus and Volcano . . . . . 9
3.3 Cascades and Columbia . . . . . 9
3.3.1 Lower bound group pruning . . . . . 10
3.3.2 Global epsilon pruning . . . . . 10
3.4 Duplicate-free Join Enumeration . . . . . 10
3.4.1 Transformation-based enumeration . . . . . 10
3.4.2 Duplicate checking in Volcano . . . . . 11
3.4.3 Duplicate-free join enumeration . . . . . 11
3.4.4 Extension to Other Rules . . . . . 11
3.4.5 Incremental Enumeration . . . . . 12

**4 Survey Of Parallel Computing Frameworks** **13**
4.1 MapReduce . . . . . 13
4.1.1 Execution model of MapReduce . . . . . 14
4.2 Hyracks . . . . . 14
4.2.1 Hyracks operators and connectors . . . . . 14
4.2.2 Execution model of Hyracks . . . . . 15
4.2.3 Algebricks . . . . . 15
4.3 Pregel . . . . . 16
4.4 Tenzing . . . . . 17
4.5 Cost Model For Parallel Computing . . . . . 18
4.5.1 Attribute Value Independence Assumption . . . . . 19
4.5.2 Costs associated with distributed processing . . . . . 19

**5 Optimization In Parallel Data Frameworks** **20**
5.1 Starfish . . . 20
5.1.1 Profiling . . . 21
5.1.2 What-if Engine . . . 21
5.1.3 Optimization using What-if Analysis . . . 21
5.2 FlumeJava . . . 22
5.2.1 Core abstractions . . . 22
5.2.2 Deferred evaluation . . . 22
5.2.3 Optimizations . . . 23
5.3 Stubby . . . 24
5.3.1 Transformations that define the plan space . . . 24

**6 Intelligent Search For Optimal Plans** **26**
6.1 Prioritization of transformations . . . 26
6.2 A-star search in query optimization . . . 26

**7 Future Work** **28**

**8 Conclusion** **29**

# Chapter 1

# Introduction

The explosion of engaging applications on the Web, and the proliferation of data intensive hardware and technologies, has resulted in an enormous increase in the amount of data generated on the Web every day. The demand for systems that can store, analyze and act upon these vast amounts of data is unprecedented. Distributed systems that employ clusters of commodity hardware computers and run robust distributed processing algorithms have provably shown to be well-suited for this task [9].

While query optimization for parallel computing systems is a relatively new field, query optimization in traditional database systems is an area of research spanning more than four decades [8]. From System R [2] and Starburst [18], to Exodus [11] and Volcano [12], to Cascades [10] and Columbia [20], each new generation of optimizers has brought about significant innovations to the field. Although there are a lot of open research problems in query optimization, current optimizers are quite advanced, usually enabling a factor of ten improvement in the execution time over an unoptimized query plan.

However, there are many differences between executing queries on single CPU systems and parallel computing systems, which ask for novel approaches to the problem of distributed query optimization. The questions of incorporating network costs, partitioning the data and scheduling the operators among different machines while selecting the optimal plan are largely unexplored. Addressing these issues should lead to a jump in the performance of the parallel processing system.

As the space of possible plans for executing a query on the parallel database is enormous, complete enumeration of all plans and selecting the best one is rendered computationally infeasible. Techniques for intelligently expanding the most promising plans and aggressively pruning the search space should benefit the efficiency of the optimizer.

## 1.1 Outline

In chapter 2 we introduce general concepts in traditional query optimization. In chapter 3 we present a survey of optimizers for the traditional database setting. It includes the Volcano based optimizer Pyro [19] developed at IIT Bombay. We also describe a technique for duplicate-free enumeration of join query transformations. In chapter 4 we discuss distributed data processing frameworks. This chapter also includes the Algebricks [5] framework which is a part of the Hyracks stack. We discuss cost models for query optimization in distributed setting. In chapter 5 we discuss certain optimization frameworks for distributed data processing. In chapter 6 we discuss literature on intelligently searching through the space of equivalent expressions during optimization. In Chapter 7 we look into possible future steps in the direction

of cost-based optimization techniques for distributed data frameworks. Then we conclude the report in Chapter 8.

# Chapter 2

# Query Optimization Overview

Query optimization is probably the most critical component of modern database systems, since the ubiquity of the database system depends on the efficiency with which a user's queries are executed by the system. As queries submitted to the database system by the user using a high-level declarative language do not specify an execution plan, it is the job of the optimizer to find a good plan, and query optimization can achieve substantial improvements in runtime and resource usage during execution of the query.

Traditionally, query optimizers follow a three step approach [19]:

- Generate all execution plans that are logically equivalent to the user-submitted query.
- Estimate the cost of executing each of the alternate plans.
- Search through the space of all generated plans to find the plan with least cost.

The three goals may not necessarily be distinct phases in an optimization scheme, but all optimizers perform these tasks during the optimization process.

## 2.1 Space of Logically Equivalent Plans

All plans that compute the same result, having the same properties, as that computed by the original query plan, are called to be logically equivalent to the given query.

The parser parses the query to generate a query tree. From the initial query tree, the space of all logically equivalent plans can be generated using transformation rules.

### 2.1.1 Logical plan space

The initial query tree consists of logical operators which are like the relational operators select, project, join, etc, and which do not specify the physical algorithm to be used during actual computation. All logical plans that are equivalent to the initial query plan constitute the logical plan space of the query.

The logical plan space can be generated by applying logical transformation rules to operators in the logical plan of the query. The new plans generated are further transformed whenever possible, until no new plans can be generated using the given set of rules.

### 2.1.2 Physical plan space

The logical plan must be converted to a physical plan than can be executed on the database system. The implementation rules define which logical operator may be implemented using which physical algorithm.

The physical plan space is generated by applying the implementation rules to all plans in the logical plan space.

## 2.2 Cost Model

The cost of executing a particular physical plan must be estimated for the purpose of finding the optimal plan. As the actual cost is not available during optimization, it is imperative for the estimates to be close to the actual costs in order to find the best plan.

The cost model defines how a cost is to be estimated for a particular operator when executed on the database. The optimizer makes use of database statistics, like histograms, and various other rules, for estimatin the cost.

## 2.3 Search For Optimal Plan

The search strategy forms an important part of the optimization scheme. As there are exponentially many alternative plans to be considered, it is infeasible to enumerate all the plans and pick the one with the lowest cost. The optimizer must incorporate pruning techniques to reduce the size of the search space that will be explored during optimization.

# Chapter 3

# Survey Of Traditional Optimizers

Query optimizers primarily deal with the question of efficiency and extensibility. As database systems evolve, new generations of optimizers are required to tackle the novel challenges brought forth by the new generations of database systems.

This chapter describes the history of query optimizers for traditional database systems [20]. The move from simple relational data models to complicated non-relational data models brought about a significant change in query optimizer technology. We believe that a similar upheaval is required to support the move from single processor to distributed data processing frameworks.

## 3.1 System R and Starburst

IBM's System R [2] query optimizer has been the foundation for many subsequent relational query optimizers. System R introduced cost-based optimization using statistics about relations and indexes stored in the system catalog to estimate the cost of execution of a query plan. The DMBS' system calatog maintains statistics about various parameters of the input relations, such as cardinality, number of pages, and available indexes. System R defined a series of formulae for estimating the size of output of an intermediate operator in the plan.

System R also contributed the bottom-up dynamic programming strategy. To find the optimal join order of multiple relations, it first computes the best plan for joining all smaller groups of the relations. It is bottom-up as it optimizes the lower level expressions first, storing the plan with the least cost for each group of relations.

However, the dynamic programming algorithm needs to consider $O(3^N)$ expressions, which is unacceptable when N is large. So the System R optimizer uses heuristics such as delaying optimization of Cartesian products until final processing, or considering only left deep trees. However, this does not guarantee optimality.

IBM's Starburst [18] is more extensible and efficient than System R. In Starburst, query optimization is done in two stages. First, the query expressed as a Query Graph Model (QGM) is passed to the QGM optimizer, which uses production rules to transform the QGM heuristically into semantically equivalent "better" QGM's. This is done to eliminate redundancy and derive expressions that are easier to optimize in a cost-based manner. The next phase is a select-project-join optimizer which consists of a modular join enumerator and a plan generator. The plan generator uses grammar- like parametrized production rules to construct access plans for joins. It constructs the optimal plan in a bottom-up fashion similar to System R.

The Starburst optimizer is efficient due to the use of sophisticated heuristics. However, the heuristics are based only on logical information and not on cost estimates, and are hard to extend to complicated queries containing non-relational operators. The grammar-like rule

based approach to transform QGMs is a step forward in extensible query optimization, but its utility for complicated queries and non-relational operators is unclear.

## 3.2 Exodus and Volcano

Exodus [11] provides a top-down optimization framework, which separates the search strategy from the data model, and also separates transformation rules and logical operators from implementation rules and physical operators. The input to the optimizer generator is a model description file which encoded the set of operators, transformation rules, implementation rules and cost model. The generator transforms the model file into a C program that is compiled and linked with the set of C procedures to generate a data model specific optimizer.

The generated optimizer transforms the query tree using the transformation rules, and maintains information about all explored alternatives in a data structure called MESH. The set of possible next transformations at any step are stored in a queue structure called OPEN. The optimizer will select a transformation from OPEN, apply the transformation on the correct nodes in MESH, do the cost estimation for new nodes, and add the newly enabled transformations to OPEN.

The Volcano optimizer generator [12] was built upon Exodus, with the goal to achieve more efficiency and extensibility. It made use of directed dynamic programming by combining dynamic programming with directed search based on physical properties, brach-and-bound pruning and heuristic guidance. In Volcano, only those sub expressions that seem to participate in promising larger plans are considered for optimization. The physical properties used for directing the search are a generalization of "interesting sort orders" in System R. The search algorithm is efficient as sub expressions are optimized only if warranted.

In Volcano, the cost model as well as logical and physical operators are encapsulated in abstract data types, making it more extensible. Also, Volcano permits exhaustive search and pruning is done at the discretion of the database implementor.

The Pyro optimizer generator developed at IIT Bombay is based on Volcano.

## 3.3 Cascades and Columbia

The Cascades optimizer generator [10] overcomes many shortcomings of Exodus and Volcano. The optimizer algorithm in Cascades adheres to the Object Oriented Paradigm, with the algorithm broken down into "task" objects which perform the optimization. The tasks are collected in a LIFO stack data structure, and a task is scheduled by popping it from the stack and invoking the "perform" function, which may result in more tasks being placed on the stack.

The optimization process is triggered by a task to optimize the top group of the initial search space, which consists of the original query. This triggers optimization of smaller groups, and new tasks are added to the stack, and new groups and expressions are added to the search space. When the top group is optimized, the best plan for the top group can be found, and hence query optimization is done.

The search strategy of Cascades differs from Volcano as Cascades only explores a group on demand, while Volcano exhaustively generates all logically equivalent expressions before the actual optimization begins. In Cascades, a group is explored on demand to create members that match a given pattern. This is efficient as it only explores expressions matching useful patterns.

The ordering of transformations to be applied is based on a "promise" value assigned to each task, which allows for intelligent prioritization of moves at each step of the optimization.

However, this feature is currently unused as most optimizer implementors assign a common promise value to all logical transformations, and a common but higher promise value to physical transformations. We discuss this in the later section of Intelligent Search For Optimal Plans.

The Columbia optimizer generator [20] is based on Cascades, with a few important modifications in the search strategy.

### 3.3.1 Lower bound group pruning

The memoized costs of some high-level plans can serve as upper bounds for subsequent optimizations. In many cases, these upper bounds could be used to avoid generating entire groups of logically equivalent expressions, called group pruning.

In Columbia, there is a group lower bound associated with a group of logically equivalent expressions, which represents the minimal cost of copying out tuples of the group and fetching tuples from the tables of the group. This group lower bound is calculate and stored in the group before the optimization of an expression as it is based only on the logical properties of the group. This improves the lower bound of the cost of an expression and consequently improves the likelihood of group pruning.

### 3.3.2 Global epsilon pruning

A plan is considered final winner of a group if its cost is within "epsilon" of the cost of an optimal plan, and no further optimization is done for that group. This technique is called Global Epsilon pruning since the epsilon is used globally during optimization of all groups.

The distance of the chosen plan from absolute optimality is bounded. If the absolute optimum has N nodes whose cost is less than the global epsilon value GE, then the cost of the chosen plan differs from the optimum by at most N * GE.

## 3.4 Duplicate-free Join Enumeration

Pellenkoft, et al [17] describe a scheme for efficiently generating logical plans from a set of transformation rules that prevents duplicates.

### 3.4.1 Transformation-based enumeration

Traditional query optimizers perform three distinct tasks while optimizing a particular query: (i) enumeration of the space of equivalent logical plans, (ii) cost estimation of alternate plans, and (iii) search for the plan with lowest cost. The query optimizers based on the Volcano optimizer generator depend on a set of transformation rules for enumeration of the entire search space.

A logical plan (or logical expression) can be transformed into equivalent logical plans by application of transformation rules. However this may lead to plans that were previously generated by another set of transformations. Current optimizers follow the approach of generating equivalent plans and then checking whether the plan was previously generated. This requires lookup in a data structure which turns out to be expensive when aggregated over multiple transformation steps during the optimization of a particular query.

A better approach should be to carefully restrict which transformation rules can be applied to a logical plan, depending on its derivation history, in order to eliminate the possibility of generating duplicates.

### 3.4.2 Duplicate checking in Volcano

The Volcano optimizer generator supports prevention of duplicate generation in a restricted sense. It supports definition of transformation rules in a way that prevents the rule being applied to the result of an earlier application of the same rule. For example, the JOIN commutativity rule can be expressed as:

%trans_rule(JOIN ?op_arg1 (?1 ?2)) ->! (JOIN ?op_arg2(?2 ?1))

An arrow of the form '->!' informs the optimizer to not apply this rule to a logical expression that was generated using this rule. This prevents 2-length cycles of transformations that may generate duplicates.

Pellenkoft, et al[17] showed that elimination of 2-length cycles provides a merely 2% performance improvement. Most of the duplicates are generated by longer cycles.

### 3.4.3 Duplicate-free join enumeration

The idea is to keep track of which rules are applicable on each operator of the logical plan, using a bitmap. A rule specifies which of the rules are allowed on the logical plans that result from the application of that rule.

For example, to generate all alternative bushy join trees for completely connected query graphs we use the following transformation rules:

**R1: Commutativity** $x\theta_0 y \rightarrow y\theta_1 x$
Disable all the rules R1, R2, R3, R4 for application on the new operator $\theta_1$

**R2: Right Associativity** $(x\theta_0 y)\theta_1 z \rightarrow x\theta_2(y\theta_3 z)$
Disable the rules R2, R3, R4 for application on the new operator $\theta_2$

**R3: Left Associativity** $x\theta_0(y\theta_1 z) \rightarrow (x\theta_2 y)\theta_3 z$
Disable the rules R2, R3, R4 for application on the new operator $\theta_3$

**R4: Exchange** $(w\theta_0 x)\theta_1(y\theta_2 z) \rightarrow (w\theta_3 y)\theta_4(x\theta_5 z)$
Disable all the rules R1, R2, R3, R4 for application on the new operator $\theta_4$

The paper provides proofs to illustrate that no duplicates are generated when the rules are applied as described, and for completely connected queries the rules R1 to R4 generate all valid bushy join orders. Similarly we can construct a set of rules for generating all linear join trees without duplicates.

The bitmap data structure is quite efficient for checking which rules to apply to each generated plan. In fact, for moderately sized queries, the performance improvement is reported as 5-6 times. Clearly, the problem of generating equivalent plans without duplicates is reduced to finding a set of rules which cover all possible plans, but disallow duplicates. As the transformation rules become complex, it becomes difficult to formulate a set of rules with the desired characteristics.

### 3.4.4 Extension to Other Rules

It would be interesting to extend the approach of Pellenkoft, et al[17] to other complicated transformation rules in a distributed framework. The challenge lies in formulating a similar set

of rules with restrictions which avoid generation of duplicates, but provably explores all possible logically equivalent expressions.

Please see the future work section.

### 3.4.5 Incremental Enumeration

Having a bitmap structure for each operator, denoting which transformation rules are applicable, allows us to perform incremental enumeration. Instead of applying all available rules at each operator, we choose to apply only one (or a few) rule(s) at a time to each operator in the plan. We find the minimum cost plan obtained from the subspace that we have enumerated. If this cost is not satisfactory, we may introduce another rule and revisit all operators with that rule. The bitmap is preserved across visits to each operator, thus we can keep track of which rules are applicable to each operator, and prevent duplicate generation across different passes over the logical plan.

This strategy should turn out to be quite useful in the scenario when a subset of the transformation rules are costly, and we may choose to ignore them if we can find a suitable plan by the application of cheaper transformation rules.

The Columbia optimizer generator maintains a bitmap for each logical expression in the search space, keeping track of which rules are allowed for that expression. However, it would only avoid 2-cycles as prevention of cycles of greater length requires a restricted set of transformation rules as described in this section.

# Chapter 4

# Survey Of Parallel Computing Frameworks

The rise Big Data and associated demand for fast processing of analytical queries on large amounts of data have led to the increasing use of parallel computing frameworks, which improve processing and input/output speeds by using multiple CPUs and disks in parallel. Centralized and clientserver database systems are not powerful enough to handle such applications.

A parallel database system seeks to improve performance through parallelization of various operations, such as loading data, building indexes and evaluating queries. Although data may be stored in a distributed fashion, the distribution is governed solely by performance considerations. This section provides an overview of various parallel data processing frameworks.

## 4.1 MapReduce

The MapReduce [9] programming model introduced by Google has become the most popular framework for distributed computation. A MapReduce program is written as a combination of a Map phase and a Reduce phase, which are inspired from the functional programming constructs of (i) map: apply a function to each data key-value pair, outputting zero or more intermediate key-value pairs, and (ii) reduce: combine the list of all intermediate values associated with a particular key to obtain a key-value pair for each distinct key.

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

For an example consider the Algorithm 1 which shows how to create an inverted index that keeps track of words in a given set of documents.

```
map(Document ID, Document) :
    for each word in Document :
        return <word, Document ID>
```

```
reduce(word, list(Document ID))
    return <word, list(Document ID)>
```

Algorithm 1: Algorithm for Constructing Inverted Index

MapReduce is convenient as the users have to provide only the Map and Reduce functions, while the run-time system takes care of parallelization of tasks, scheduling tasks on separate

machines, and redistribution of data between tasks. MapReduce encapsulates the intricacies of distributed processing from the user, allowing for robust and easy execution of distributed programs.

### 4.1.1 Execution model of MapReduce

The execution of a job in MapReduce framework goes in the following manner [9] :

- Input data is automatically partitioned into $M$ partitions if not already partitioned in a distributed file system.
- User defined map function is invoked in parallel in each of these $M$ different machines.
- Intermediate key-value pair produced by map are written to local disk, partitioned into $R$ regions by some partitioning function. Here $R$ corresponds to the number of reduce tasks.
- Next the $R$ reducer machines are notified about the addresses of relevant partitions.
- The reducer machines fetches the data from mapper machines. Here all the data corresponding to a particular intermediate key will go to a single reduce machine.
- Reducer machine sorts the data by the intermediate keys.
- Reducer machine then iterates over the sorted data and send the set of all the values corresponding to a particular key to the user defined reduce function along with the corresponding key.
- Output of the $R$ reducer will be the final output of the Map-Reduce program.

## 4.2 Hyracks

The MapReduce model constrains the programmer to the subspace of distributed programs that can be written as a sequence of Map and Reduce tasks. Hyracks [4] is an open-source framework which provides more flexibility to the user by supporting various operators and accepting input programs as a directed acyclic graph, with nodes representing operators and edges representing redistribution of data.

### 4.2.1 Hyracks operators and connectors

Hyracks comes with a library of operators and connectors which the end-users can use to assemble their programs. Hyracks also supports extension of the library by inclusion of user-defined operators and connectors.

1) Operators:

- File Readers/Writers: Operators to read and write files in various formats from/to local file systems and the Hadoop Distributed Filesystem.
- Mappers: Operators to evaluate a user-defined function on each item in the input.
- Sorters: Operators that sort input records using user- provided comparator functions.
- Joiners: Binary-input operators that perform equi-joins. We have implementations of the Hybrid Hash-Join and Grace Hash-Join algorithms.

- Aggregators: Operators to perform aggregation using a user-defined aggregation function. We have implemented a hash-based grouping aggregator and also a pre-clustered version of aggregation.

2) Connectors: Connectors distribute data produced by a set of sender operators to a set of receiver operators.

- M:N Hash-Partitioner: Hashes every tuple produced by senders (on a specified set of fields) to generate the receiver number to which the tuple is sent. Tuples produced by the same sender keep their initial order on the receiver side.
- M:N Hash-Partitioning Merger: Takes as input sorted streams of data and hashes each tuple to find the receiver. On the receiver side, it merges streams coming from different senders based on a given comparator, thus producing ordered partitions.
- M:N Range-Partitioner: Partitions data using a speci- fied field in the input and a range-vector.
- M:N Replicator: Copies the data produced by every sender to every receiver operator.
- 1:1 Connector: Connects exactly one sender to one receiver operator.

### 4.2.2 Execution model of Hyracks

Execution of a job in Hyracks goes as follows [4]:

- Hyracks accepts jobs in the form of directed acyclic graph where the nodes represent operators and edges represent redistribution of data.
- Operators are first expanded into their constituent activities to create an Activity Graph from the input DAG. For example a nested join operator is made up of build and probe activities.
- The activities will expose the blocking characteristics of the operators. This will provide important scheduling information to the system.
- At runtime Hyracks analyzes the Activity Graph to infer collection of activities that can be executed simultaneously and calls them Stages.
- In the final plan Stages are ordered in the sequence they becomes ready to execute.

### 4.2.3 Algebricks

The Hyracks software stack (Figure 4.1) contains an algebra layer which translates high-level declarative language programs into Hyracks job specifications that can be executed on the Hyracks system layer. Algebricks performs query processing and optimization and is designed in to be easily extended to support new high-level declarative languages.

Algebricks consists of the following parts [5]:

- A set of logical operators,
- A set of physical operators,
- A rewrite rule framework,

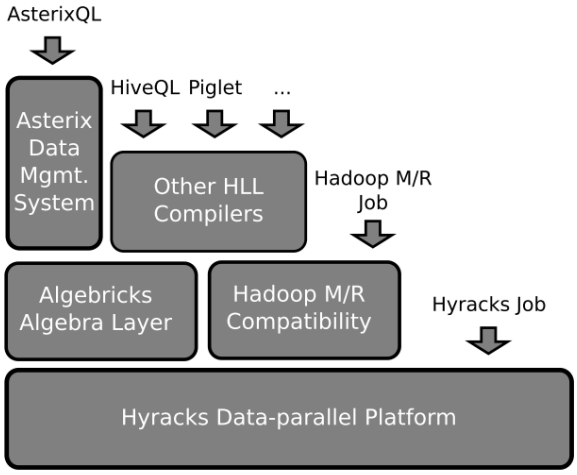


Figure 4.1: The ASTERIX stack [4]

- A set of generally applicable rewrite rules,
- A metadata provider API that exposes metadata (catalog) information to Algebricks, and,
- A mapping of physical operators to the runtime operators in Hyracks.

Algebricks provides all of the traditional relational operators such as select, project, and join. In addition, Algebricks enables the expression of correlated queries through the use of a subplan operator. The groupby operator in Algebricks allows complete nested plans to be applied to each group.

The Algebricks optimizer uses Logical-to-Logical rewrite rules to create alternate logical formulations of the initially provided DAG. Logical-to-Physical rewrite rules then generate a DAG of physical operators that specify the algorithms to use to evaluate the query. For example, a join operator might be rewritten to a hash-join physical operator. This process of rewriting into a physical DAG uses partitioning properties and local physical properties of input data sources. Data exchange operators are used to perform redistribution of data between partitions.

The Algebricks toolkit contains a rewriting framework that allows users to write their own rewrite rules but also comes with a number of out of the box rules that the user can choose to reuse for compiling their high-level language. Examples of rules that seem likely to apply for most languages include:

- Push Selects: Rule for pushing filters lower in the plan to eliminate data that is not useful to the query.
- Introducing Projects: Rule to limit the width of intermedi- ate tuples by eliminating values that are no longer needed in the plan.
- Query Decorrelation: Rule to decorrelate nested queries to use joins when possible.

## 4.3 Pregel

Google's Pregel [16] provides a simple solution to large-scale graph processing problems. It facilitates synchronized round-based computation at vertices with message passing between rounds. Pregel avoids the communication overheads and programming complexity incurred by

iterative MapReduce computations, as it keeps vertices and edges on the machines that perform the computation, while using network transfers only for messages.

In an iteration (superstep S), a vertex V receives messages sent to it in the previous iteration (S-1), sends messages that will be received in the next step (S+1), modifies its own state and the state of its outgoing edges, and mutates the graphs topology. This synchronized superstep model is inspired by Valiant's Bulk Synchronous Parallel model.

The user program begins executing on a cluster of machines over the partitioned graph data. One of the machines acts as the master and coordinates worker activity. The master determines how many partitions the graph will have, and assigns one or more partitions to each worker machine. The number may be controlled by the user. Each worker is also given the complete set of assignments for all workers so that the worker knows which other worker to enqueue messages for its outgoing edges. Fault-tolerance is achieved by checkpointing and replaying on machine failure. However, a self-stabilizing graph algorithm can be executed without fault-tolerance and finished early.

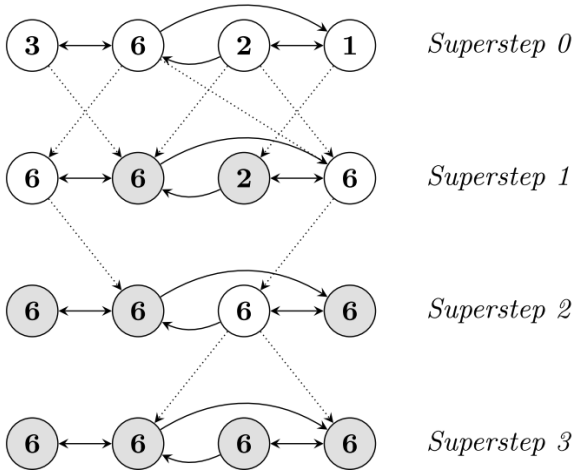


Figure 4.2: Pregel's Maximum Value Example [16]

Figure 4.2 illustrates these concepts using a simple example [16]: given a strongly connected graph where each vertex contains a value, it propagates the largest value to every vertex. In each superstep, any vertex that has learned a larger value from its messages sends it to all its neighbors. When no further vertices change in a superstep, the algorithm terminates.

## 4.4 Tenzing

Tenzing [7] is a query engine built on top of MapReduce for ad hoc analysis of Google data. Tenzing supports a mostly complete SQL implementation (with several extensions) combined with several key characteristics such as heterogeneity, high performance, scalability, reliability, metadata awareness, low latency, support for columnar storage and structured data, and easy extensibility. Tenzing is currently used internally at Google by 1000+ employees and serves 10000+ queries per day over 1.5 petabytes of compressed data.

A typical Tenzing query goes through the following steps [7]:

1. A user (or another process) submits the query to the query server through the Web UI, CLI or API.

2. The query server parses the query into an intermediate parse tree.
3. The query server fetches the required metadata from the metadata server to create a more complete intermediate format.
4. The optimizer goes through the intermediate format and applies various optimizations.
5. The optimized execution plan consists of one or more MapReduces. For each MapReduce, the query server finds an available master using the master watcher and submits the query to it. At this stage, the execution has been physically partitioned into multiple units of work(i.e. shards).
6. Idle workers poll the masters for available work. Reduce workers write their results to an intermediate storage.
7. The query server monitors the intermediate area for results being created and gathers them as they arrive. The results are then streamed to the upstream client.

Tenzing supports almost all SQL92 standard and some extensions from SQL99:

- projection and filtering (Tenzing can do some optimizations for some of these and depending on the data source )
- set operations (implemented in the reduce phase)
- nested queries and subqueries
- aggregation and statistical functions
- analytic functions (syntax similar to PostgreSQL/Oracle)
- OLAP extensions
- JOINs

The Tenzing paper demonstrated that it is possible to create a functional SQL engine on top of the MapReduce framework. With relatively minor enhancements to the MapReduce framework, it is possible to implement a large number of optimizations currently available in commercial database systems, and create a system which can compete with commercial MPP DBMS in terms of throughput and latency.

## 4.5 Cost Model For Parallel Computing

An estimate of the cost of executing a query can be obtained from a model of the costs of various operations involved in the query execution. As the efficiency of the optimizer is proportional to how accurately the estimated costs reflect the true cost of running the query, improvements in the cost model used by the optimizer can have a significant effect on the overall performance.

It is difficult to accurately predict the cost of running a query, hence the cost model usually takes simplifying assumptions, like the attribute value independence assumption.

### 4.5.1 Attribute Value Independence Assumption

For a query involving two or more attributes of the same relation, its result size depends on the joint data distribution of those attributes; i.e., the frequencies of all combinations of attribute values in the database. Due to the multi-dimensional nature of these distributions and the large number of such attribute value combinations, direct approximation of joint distributions can be rather complex and expensive. In practice, most commercial DBMSs adopt the attribute value independence assumption. Under this assumption, the data distributions of individual attributes in a relation are independent of each other and the joint data distribution can be derived from the individual distributions (which are approximated by one-dimensional histograms).

### 4.5.2 Costs associated with distributed processing

The cost model for a query running on a distributed processing framework must incorporate additional costs like network costs, data redistribution costs, etc. which are not considered in traditional cost models.

We plan to implement a cost model for queries on Hyracks, which will consider all such costs during query optimization (see future work, section 7).

# Chapter 5

# Optimization In Parallel Data Frameworks

This chapter presents a survey of optimization techniques in distributed data frameworks.

## 5.1 Starfish

A typical MapReduce engine, like Hadoop [1], has hundreds of configuration parameters which control the execution of the program. The work of Herodotou et al. [3, 13] shows a way of cost based optimization by choosing optimal configuration parameters that affect the performance significantly.

Hadoop has more than 190 configuration parameters out of which 10-20 parameters can have significant impact on the job performance. Table 5.1 lists some of the crucial parameters.

| | |
|---|---|
| *io.sort.mb* | Buffer size for storing and sorting at a particular map and reduce task |
| *io.sort.spill.percent* | Threshold of map-side memory buffer to trigger a sort and spill of the key-value pair |
| *mapreduce.combine.class* | The optional Combiner function to preaggregate map output before transferring to reduce task |
| *min.num.spills.for.combine* | Minimum number of spill files to trigger the use of combiner on the map output |
| *mapred.reduce.tasks* | Number of reduce tasks |
| *io.sort.factor* | Number of sorted streams to merge at once during multiple phase external sort |

Table 5.1: Some of the significant configuration parameters of Hadoop

Due to the black-box nature of map and reduce functions (as they are written by user and not known to the system in some programming language like C++, Java, Python ), it is very hard to automatically find optimal values of the parameters. Besides this, the lack of schema and statistics about the input data poses another challenge for cost based optimization. The key and values are often extracted from input data at run-time, so it is not possible to have the statistics beforehand.

A profiler first collects statistical information of the map-reduce program, then the What-If engine estimates the cost for a given input data, resource cluster and configuration parameters.

Lastly the cost-based optimizer chooses optimal configuration parameters.

### 5.1.1 Profiling

The Starfish framework is responsible for invoking map, reduce and other functions in the user program. This property is used by profiler to monitor and collect run-time data corresponding to various dataflow and cost fields and statistics.

Data-flow fields like *Number of map tasks in the job*, *Map output size*, *Number of spills*, *Number of records in spill file*, *Spill file size* depend only on input data and configuration parameters. Cost fields like *Map phase time in the map task*, *Spill phase time in the map task*, *Merge phase time in map/reduce task* depend on input data, resource cluster as well as configuration parameters. Data-flow statistics like *Map selectivity*, *Reducer selectivity*, *Combiner selectivity* depend only on input data. Cost statistics fields like *I/O cost for reading/writing per byte* and *CPU cost for a phase per record/byte* depend on the clusters.

The profiler applies dynamic instrumentation by specifying a set of *event-condition-action (ECA)* rules where the events are entry or exit of function calls, memory allocation and system calls. The actions involve getting the duration of a function call, examining the memory state, or counting the number of bytes transferred. A valuable feature of dynamic instrumentation is that it can be turned on and off seamlessly to generate approximate job profile while keeping run-time overhead low.

### 5.1.2 What-if Engine

What-if engine answers what-if questions of the following form [13]:

> Given the profile of a job $j =< p, d_1, r_1, c_1 >$ that runs a MapReduce program $p$ over input data $d_1$ and cluster resources $r_1$ using configuration $c_1$, what will the performance of program $p$ be if $p$ is run over input data $d_2$ and cluster resources $r_2$ using configuration $c_2$? That is, how will job $j^{'} =< p, d_2, r_2, c_2 >$ perform?

The What-if engine takes into account the size and block layout of the input data $d$ and number of nodes, network topology, map-reduce slots per node and memory available of a cluster resource $r$.

The engine estimates a virtual job profile for the hypothetical job $j^{'}$. As the Map-Reduce framework lacks the structured data representation and histogram-like statistics of a database system, the engine assumes that the data-flow statistics will be the same as the given job profile $j$. It computes data-flow fields proportional to its input data size. For cost fields, it either does similar proportional estimation as the data-flow fields or uses a trained black-box model for a cluster $r_2$ that differs significantly from given cluster $r_1$.

Now the engine uses a *Task Scheduler Simulator* to simulate the scheduling and execution of map and reduce task of job $j^{'}$ and estimates its performance for the given configuration parameters.

### 5.1.3 Optimization using What-if Analysis

The cost-based optimizer is used to find the optimal values of the configuration parameters by searching the space of configuration parameters and making what-if calls for each parameter setting.

To efficiently search the high dimensional space $S$ of configuration parameters, the individual parameters in $c$ are grouped into a cluster $c^{(i)}$. Now to find the optimal cluster $c^{(i)}_{opt}$ the search space $S$ is searched in either of the following techniques:

- **Griding (Equispaced or Random) :** Here, the domain $dom(c_i)$ of each configuration parameter is discretized in to $k$ values. The values may be equispaced or randomly chosen. The CBO makes call to what-if engine for each of the $k^n$ settings and selects the settings with lowest estimated execution time.
- **Recursive Random Search (RRS) :** RRS first samples the subspace randomly to identify promising regions that contain the optimal settings with high probability. It then recursively samples these spaces which either move or shrink gradually to local optimal settings.

## 5.2 FlumeJava

FlumeJava [6] is a pure Java library that provides special Java collections and operations which get translated into MapReduce jobs. One of the primary advantages is transparently chaining together multiple MapReduce jobs into a processing pipeline, which serves a similar purpose as Pig. FlumeJava performs optimizations on the resulting dataflow graph to combine and optimize the different MapReduce stages, but still has to deal with ultimately reading from and writing to disk between stages (unlike DryadLINQ, which supports in-memory transfer).

The resulting pipeline comes close to the performance of a hand-optimized MapReduce pipeline.

### 5.2.1 Core abstractions

The central class of the FlumeJava library is PCollection<T>, a (possibly huge) immutable bag of elements of type T. A PCollection can either have a well-defined order (called a sequence), or the elements can be unordered (called a collection). Data sets represented by multiple file shards can be read in as a single logical PCollection. A second core class is PTable<K,V>, which represents a (possibly huge) immutable multi-map with keys of type K and values of type V. PTable<K,V>is a subclass of PCollection<Pair<K,V>>, and indeed is just an unordered bag of pairs.

The main way to manipulate a PCollection is to invoke a data-parallel operation on it. The FlumeJava library defines only a few primitive data-parallel operations; other operations are implemented in terms of these primitives. The core data-parallel primitive is parallelDo(), which supports elementwise computation over an input PCollection<T>to produce a new output PCollection<S>. This operation takes as its main argument a DoFn<T, S>, a function-like object defining how to map each value in the input PCollection<T>into zero or more values to appear in the output PCollection<S>. parallelDo() can be used to express both the map and reduce parts of MapReduce.

### 5.2.2 Deferred evaluation

In order to enable optimization as described in the next section, FlumeJavas parallel operations are executed lazily using deferred evaluation. Each PCollection object is represented internally either in deferred (not yet computed) or materialized (computed) state. A deferred PCollection holds a pointer to the deferred operation that computes it. A deferred operation, in turn, holds references to the PCollections that are its arguments (which may themselves be

deferred or materialized) and the deferred PCollections that are its results. When a FlumeJava operation like parallelDo() is called, it just creates a ParallelDo deferred operation object and returns a new deferred PCollection that points to it. The result of executing a series of FlumeJava operations is thus a directed acyclic graph of deferred PCollections and operations; we call this graph the execution plan.

To actually trigger evaluation of a series of parallel operations, the user follows them with a call to FlumeJava.run(). This first optimizes the execution plan and then visits each of the deferred operations in the optimized plan, in forward topological order, and evaluates them. When a deferred operation is evaluated, it converts its result PCollection into a materialized state, e.g., as an in-memory data structure or as a reference to a temporary intermediate file. FlumeJava automatically deletes any temporary intermediate files it creates when they are no longer needed by later operations in the execution plan.

### 5.2.3 Optimizations

The FlumeJava optimizer transforms a user-constructed, modular FlumeJava execution plan into one that can be executed efficiently. The optimizer is written as a series of independent graph transformations.

One of the simplest and most intuitive optimizations is ParallelDo producer-consumer fusion (Figure 5.1), which is essentially function composition or loop fusion. If one ParallelDo operation performs function f, and its result is consumed by another ParallelDo operation that performs function g, the two ParallelDo operations are replaced by a single multi-output ParallelDo that computes both f and g o f. If the result of the f ParallelDo is not needed by other operations in the graph, fusion has rendered it unnecessary, and the code to produce it is removed as dead.

ParallelDo sibling fusion applies when two or more ParallelDo operations read the same input PCollection. They are fused into a single multi-output ParallelDo operation that computes the results of all the fused operations in a single pass over the input.

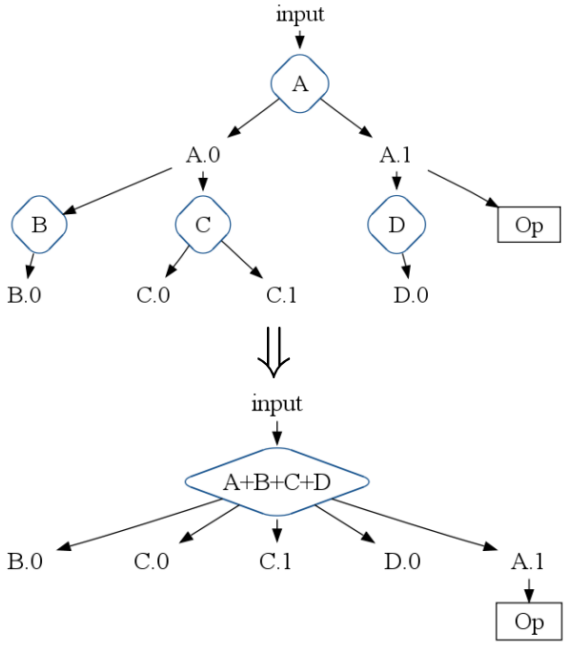


Figure 5.1: ParallelDo Producer-Consumer and Sibling Fusion [6]

## 5.3 Stubby

Automatic cost-based optimization of MapReduce workflows remains a challenge due to the multitude of interfaces, large size of the execution plan space, and the frequent unavailability of all types of information needed for optimization. Stubby [15] is a cost-based optimizer that searches selectively through the subspace of the full plan space that can be enumerated correctly and costed based on the information available in any given setting. Stubby enumerates the plan space based on plan-to-plan transformations and an efficient search algorithm. Stubby is designed to be extensible to new interfaces and new types of optimizations, which is a desirable feature given how rapidly MapReduce systems are evolving.

Stubby accepts input in the form of an annotated MapReduce workflow and returns an equivalent, but optimized, plan. Annotations are a generic mechanism for workflow generators to convey useful information found during workflow generation. Stubby will find the best plan subject to the given annotations, while working correctly (but not optimally) when zero to few annotations are given.

A MapReduce workflow is a Directed Acyclic Graph (DAG) G that represents a set of MapReduce jobs and their producer-consumer relationships. Each vertex in G is either a MapReduce job J or a dataset D. Each edge in G is between a job (vertex) J and a dataset (vertex) D, and denotes whether D is an input or an output dataset of J.

### 5.3.1 Transformations that define the plan space

The plan space of MapReduce workflows is enumerated by applying transformations to the input workflow. Some of the transformations are:

- **Intra-job Vertical Packing Transformation**

  An intra-job vertical packing transformation converts a MapReduce job into a Map-only job. Suppose M and R respectively denote the map and reduce functions of the job. Without the vertical packing transformation, M will be invoked in the jobs map tasks, and R will be invoked in the jobs reduce tasks. After the transformation, the M and R functions will be pipelined together and invoked in the new jobs map tasks. The data output by M will now be provided directly to R without going through the partition, sort, and shuffle phases of MapReduce job execution.

- **Inter-job Vertical Packing Transformation**

  An inter-job vertical packing transformation moves functions from a job J into another job, completely eliminating the need for J.

- **Horizontal Packing Transformation**

  A horizontal packing transformation packs the map (reduce) functions of multiple jobs that read the same dataset into the same map (reduce) task of a transformed job. While vertical packing transformations pipeline functions sequentially, a horizontal packing transformation puts multiple map (reduce) functions from separate parallel pipelines into a single jobs map (reduce) task.

- **Partition Function Transformation**

  Partition function transformation changes how the map output key-value pairs are partitioned and sorted during the execution of a job. This transformation includes, but is not limited to: (i) changing the partitioning type (default is hash), (ii) changing the splitting

points for range partitioning, and (iii) changing the fields on which per-partition sorting happens.

- **Configuration Transformation**

  A configuration transformation changes the configuration of a MapReduce job in a workflow, for example, it may change the map output buffer size of a job.

# Chapter 6

# Intelligent Search For Optimal Plans

The search strategy of an optimizer plays an important part in determining its performance. An important aspect of the search strategy is determining the order in which transformation rules should be applied at each node of the query tree.

## 6.1 Prioritization of transformations

The Cascades [10] and Columbia [20] optimizer generators introduced the notion of "tasks" for optimizing expressions by applying transformation rules. The tasks have a "promise" value that can be used to prioritize between different tasks. However, this feature is unused, or used to a limited extent, in Cascades and Columbia. For e.g., in Columbia, all tasks embodying logical transformations are given a promise value of 1, while those embodying physical transformations are given a promise value of 2. This gives a higher priority to generation of physical plans than the generation of alternate logical plans during optimization, while giving equal preference to any two logical transformations or, similarly, any two physical transformations.

However, choosing the logical (or physical) transformations that lead to optimal, or close to optimal, plans before other transformations can make pruning more effective as the costlier plans will not be enumerated. This can be achieved if we assign higher promise values to the tasks that expand better equivalent plans.

The primary issue is to ascertain which transformations correspond to plans with lower costs. The answer to this question depends on a lot of parameters, for e.g. the type of operator being substituted, the transformation rule, logical and physical properties of the required context, database statistics, etc. We plan to research this approach further in later stages of this project (See Future Work, chapter 7)

## 6.2 A-star search in query optimization

One approach to this issue is to view the problem as a heuristic search for finding the optimal plan [14]. Yoo et al. [21] have discussed an intelligent search method for query optimization by semi-joins. They have used the A* search algorithm, which is popular in AI applications, to "intelligently" search through the plan space.

In the A* algorithm, the search is controlled by a heuristic function f. The state x chosen for expansion (i.e., whose immediate successors will be generated) is the one which has the smallest value f(x) among all generated states that have not been expanded so far. The purpose of f is to evaluate states in terms of their goodness in linking the initial state to the goal state. The search stops when the state chosen for expansion is the goal state. The function f considers two

components: the cost of reaching x from the initial state, and the cost of reaching the goal state from x. Accordingly, $f(x)$ is defined by

$$f(x) = g(x) + h(x)$$

where $g(x)$ estimates the minimum cost of a trajectory from the initial state to x, and $h(x)$ estimates the minimum cost from x to the goal state. Thus, the value $f(x)$ estimates the minimum cost of a solution trajectory passing through x. The function $h(x)$ represents the heuristic information that will determine the power of the algorithm. In order for A* to achieve our objectives of optimality and efficiency, its evaluation function must satisfy two conditions called admissibility and consistency.

We plan to work on devising the heuristic functions $g(x)$ and $h(x)$ that satisfy these conditions, for application of A* search in cost-based query optimization for parallel computing frameworks. (Future Work, chapter 7)

# Chapter 7

# Future Work

We have identified a few key areas in the optimization algorithm where further work can have significant impact on the efficiency and quality of the optimized query plan that is outputted by the query optimizer:

- **Intelligent Search**

  As discussed in section 6.1, prioritizing between various transformation rules applicable at each step should lead to improved performance of the search strategy of the query optimizer. This leads to the notion of intelligent search, i.e. choosing to expand those rules first that are expected to lead to optimal plans.

  This is especially important for parallel query optimization, as the search space is much larger due to a high number of feasible data partitioning schemes. In many cases, the queries tend to be quite large, rendering exhaustive enumeration of even the logical plan space quite impractical.

  It will be interesting to research which factors prove to be decisive in correctly identifying the rules that lead to optimal plans.

- **Cost model for parallel processing**

  We plan to design a cost model for parallel data processing frameworks like Hyracks, which will consider the costs of data repartitioning, network costs, etc.

- **Duplicate-free enumeration**

  The approach of Pellenkoft et al. [17] described in section 3.4 can be extended to transformation rules for distributed query optimization. A minimal set of rules which cover the entire space of logically equivalent plans, along with restrictions on the application of the rules to avoid generating duplicates, can be expected to give a significant boost in performance as the number of plans considered will reduce drastically, at no extra computations on the part of the optimizer.

- **Optimizing Java data structures**

  The data structures associated with the equivalence and operator nodes in the expanded query DAG can be optimized to save memory and time overhead. As complex queries may contain hundreds or even thousands of such nodes in the expanded DAG, any saving incurred by such an optimization will have a significant improvement in the optimizer performance. The Java data structures incur non-trivial amounts of book-keeping related overhead. Designing the data structures in an efficient manner is an important open problem in the design of query optimizers.

# Chapter 8

# Conclusion

In the report, we presented a literature survey in the area of traditional database optimization techniques. This was followed by survey of parallel database frameworks, including the popular MapReduce framework, as well as the more general Hyracks. Besides providing formal structural properties to be used by an parallel optimizer, we also described a technique for duplicate-free join enumeration and methods for intelligent search during optimization.

We presented a survey of current optimization techniques for parallel data frameworks. We described a cost model appropriate for cost-based optimization in the parallel setting, and conluded with possible future work in some of the techniques described earlier, i.e. duplicate-free transformations, cost model and intelligent search, for application in optimization of parallel database queries.

# Bibliography


[1] Apache. hadoop. `http://hadoop.apace.org`.

[2] Astrahan, M. M., Blasgen, W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., and Griffiths, P. P. System R : Relational Management Approach to Database. *ACM Transactions on Database Systems 1*, 2 (1976), 97–137.

[3] Babu, S. Towards automatic optimization of MapReduce programs. *Proceedings of the 1st ACM symposium on Cloud Computing* (2010).

[4] Borkar, V., Carey, M., and Grover, R. Hyracks: A flexible and extensible foundation for data-intensive computing. *ICDE* (2011).

[5] Borkar, V., Carey, M., and Li, C. Inside Big Data management: ogres, onions, or parfaits? *EDBT/ICDT Joint Conference* (2012).

[6] Chambers, C., Raniwala, A., and Perry, F. FlumeJava: easy, efficient data-parallel pipelines. *ACM Sigplan* (2010), 363–375.

[7] Chattopadhyay, B., Lin, L., Liu, W., and Mittal, S. Tenzing a sql implementation on the mapreduce framework. *PVLDB* (2011), 1318–1327.

[8] Chaudhuri, S. An Overview of Query Optimization.

[9] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (December 2004), pp. 137–150.

[10] Graefe, G. The cascades framework for query optimization. *Data Engineering Bulletin* (1995), 19–28.

[11] Graefe, G., and DeWitt, D. J. The EXODUS optimizer generator. *ACM SIGMOD Record 16*, 3 (Dec. 1987), 160–172.

[12] Graefe, G., and McKenna, W. The Volcano optimizer generator: extensibility and efficient search. *Proceedings of IEEE 9th International Conference on Data Engineering* (1993), 209–218.

[13] Herodotou, H., and Babu, S. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proceedings of the VLDB Endowment* (2011).

[14] Ioannidis, Y. E. Query optimization. *ACM Computing Surveys 28*, 1 (Mar. 1996), 121–123.

[15] Lim, H., Herodotou, H., and Babu, S. Stubby: a transformation-based optimizer for MapReduce workflows. *Proceedings of the VLDB Endowment 5*, 11 (2012), 1196–1207.

[16] Malewicz, G., Austern, M., and Bik, A. Pregel: a system for large-scale graph processing. *ACM SIGMOD* (2010), 135–145.

[17] Pellenkoft, A., Galindo-Legaria, C. A., and Kersten, M. L. The complexity of transformation-based join enumeration. In *VLDB* (1997), pp. 306–315.

[18] Pirahesh, H. Extensible/rule based query rewrite optimization in Starburst. *ACM SIGMOD* (1992).

[19] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference* (2000), pp. 249–260.

[20] Xu, Y. Efficiency in the columbia database query optimizer. Master's thesis, Portland State University, Portland, 1998.

[21] Yoo, H., and Lafortune, S. An intelligent search method for query optimization by semijoins. *IEEE Transactions on Knowledge and Data Engineering 1*, 2 (June 1989), 226–237.