Chapter 9 - Performance at Large Scale

Marc Jourdan, Ivan Tishchenko
January 10, 2019

1 Lecture

Sources of bottleneck:

- Memory → Memory-bound
- CPU → CPU-bound
- Disk I/O → (Disk) IO-bound: Sweet Spot for MapReduce and Spark
- Network I/O → (Network) IO-bound

Measurements:

- Latency
- Throughput
- Total response time = Latency + Transfer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute typical instruction</td>
<td>1 ns</td>
</tr>
<tr>
<td>fetch from L1 cache memory</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>branch misprediction</td>
<td>5 ns</td>
</tr>
<tr>
<td>fetch from L2 cache memory</td>
<td>7 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
</tr>
<tr>
<td>fetch from main memory</td>
<td>100 ns</td>
</tr>
<tr>
<td>send 2K bytes over 1Gbps network</td>
<td>20 μs</td>
</tr>
<tr>
<td>read 1MB sequentially from memory</td>
<td>250 μs</td>
</tr>
<tr>
<td>fetch from new disk location (seek)</td>
<td>8 ms</td>
</tr>
<tr>
<td>read 1MB sequentially from disk</td>
<td>20 ms</td>
</tr>
<tr>
<td>send packet US to Europe and back</td>
<td>150 ms</td>
</tr>
</tbody>
</table>
• Speedup = \frac{\text{Latency}_{\text{old}}}{\text{Latency}_{\text{new}}}

  – **Amdahls law**: Speedup = \frac{1}{1-p+s} - Constant problem size
  
  – **Gustafsons law**: Speedup = 1 - p + ps - Constant computing power

What can we act on?

• Number of machines (scale out)

• Machine features (CPU, Memory, ...) (scale up): easy way and hence last resort

• Code: eg. `++i` instead of `i++`, Optimize large loops if possible, Avoid exception catching, Avoid polymorphism, Avoid virtual functions, Go low level if needed

• Size of chunks: Allow some "liquidity" with more chunks (without pushing it too much)

• Storage format: Binary format instead of Raw format

• Network usage: Keep shuffling to a minimum, Push down projection and selection as close as possible to the source

  
  • Architecture

  The objective is to find a trade-off between too few executor with a lot of ressources and too many executor with few ressources.

  **Tail Latency**: in large scales, we have in reality a node taking far more time than all the other.

  \(^1_p \text{ parallelizable proportion, } s \text{ speedup on the parallelizable part} \)
Reasons:

- Resources are shared on a machine
- Resources are shared Network Globally
- Background daemon creates hiccups
- Queues
- Power limits (hyperthreading)
- Garbage collection
- Energy management

Possible improvements:

- Keep low-level queues short
- Break splits and queries further down
- Manage background activities
- Hedge requests: Task duplicates, first done wins
• Deferred hedge requests: When 95% percentile reached on a node, launch duplicate task and discard the previous one
• Tied request: Enqueue where the queue is shorter, enqueue twice (now or wait for percentile to be reached), cancel when other starts
• Micro-partitions with Selective replication: predict this will take long, assign less partitions to this node
• Latency-induced probation: We shut down a machine that is too slow
• Completely drop the last 0.01
• Canary requests: test code used for the first time on 1-2 nodes

2 Mandatory Readings

2.1 Scalability! But at what COST?

Contribution of this work: a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread.

Definition: The COST of a given platform for a given problem is the hardware configuration (e.g., number of cores, ...) required before the platform outperforms a competent single-threaded implementation.

Advantage: COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

Link to the lecture: "You can have a second computer once you've shown you know how to use the first one." Paul Barham

Striking problem in research this area: few papers directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?

Surprisingly, any system can scale arbitrarily well with a sufficient lack of care in its implementation. It is even possible to damage scalability when removing parallelizable overheads. Moreover recent graph processing papers have high (even unbounded) COST. Unbounded COST means that no configuration outperforms the best single-threaded implementation.

Evaluating method: Graph computation (data-parallel computation that is not trivially parallelized) comparison between single-threaded implementation and scaled system.
Metrics:

- **PageRank**: iteratively updates a rank maintained for each vertex. In each iteration a vertex's rank is uniformly divided among its outgoing neighbors, and then set to be the accumulation of scaled rank from incoming neighbors. A dampening factor alpha is applied to the ranks, the lost rank distributed uniformly among all nodes.

- **Connected Components**: The connected components of an undirected graph are disjoint sets of vertices such that all vertices within a set are mutually reachable from each other. In label propagation, each vertex maintains a label (initially its own ID), and iteratively updates its label to be the minimum of all its neighbors labels and its current label. The process propagates the smallest label in each component to all vertices in the component, and the iteration converges once this happens in every component.

Even dummy single-threaded implementation outperforms most of the time multi-threaded implementations. The single-threaded implementation can also be improved (improved algorithms, improved graph layout).

Aspects of scalable systems design and implementation contributing to overhead and increased COST:

- The computational model presented by the system restricts the programs one may express
- The target hardware may reflect different trade-offs, perhaps favoring capacity and throughput over high clock frequency
- The implementation of the system may add overheads a single thread doesn't require

Important to distinguish between scalability and efficient usage of resource. the implementation of the system may introduce overheads that conceal the performance benefits of a scalable system.

**Conclusion**: It is important to evaluate the COST, both to explain whether a high COST is intrinsic to the proposed system, and because it can highlight avoidable inefficiencies and thereby lead to performance improvements for the system.

### 2.2 Spark SQL: Relational Data Processing in Spark

*See below, at the end of the file*
3 Recommended Readings

3.1 The Tail at Scale

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services. It is challenging for service providers to keep the tail of latency distribution short for interactive services as the size and complexity of the system scales up or as overall use increases.

Tail Latency: Temporary high-latency episodes (unimportant in moderate-size systems) may come to dominate overall service performance at large scale.

Objective: Creating a predictably responsive whole out of less-predictable parts.

Why variability exists?

• Shared resources
• Daemons
• Global resource sharing
• Maintenance activities
• Queueing
• Power limits
• Garbage collection
• Energy management

Key insights:

• Even rare performance hiccups affect a significant fraction of all requests in large-scale distributed systems

• Eliminating all sources of latency variability in large-scale systems is impractical, especially in shared environments

• Using an approach analogous to fault-tolerant computing, tail-tolerant software techniques form a predictable whole out of less-predictable parts.
Component-Level Variability Amplified By Scale  Technique: parallelize sub-operations across many different machines, where each sub-operation is co-located with its portion of a large dataset. Parallelization happens by fanning out a request from a root to a large number of leaf servers and merging responses via a request-distribution tree. These sub-operations must all complete within a strict deadline for the service to feel responsive. Techniques that concentrate on these slow outliers can yield dramatic reductions in overall service performance.

Reducing Component Variability  Interactive response-time variability can be reduced by ensuring interactive requests are serviced in a timely manner through many small engineering decisions:

- Differentiating service classes and higher-level queuing: used to prefer scheduling requests for which a user is waiting over non-interactive requests
- Reducing head-of-line blocking: break long-running requests into a sequence of smaller requests to allow interleaving of the execution of other short-running requests
- Managing background activities and synchronized disruption.

Two solutions to deal with tail latency when it can’t be eliminated:

- Within Request Short-Term Adaptations: operate at a time scale of tens of milliseconds, before longer-term techniques have a chance to react
- Cross-Request Long-Term Adaptations: operate at a time scale of tens of seconds to minutes and are meant to mask the effect of longer-term phenomena

Within Request Short-Term Adaptations\(^2\)

- Hedged requests, especially Deferred hedged request
- Tied requests. Tied requests allow the workloads to be consolidated into a single cluster, resulting in dramatic computing cost reductions
- Alternative: probe remote queues first, then submit the request to the least-loaded server

\(^2\)Effective only when the phenomena that causes variability does not tend to simultaneously affect multiple request replicas
Cross-Request Long-Term Adaptations. A static assignment of a single partition to each machine is rarely sufficient: the performance of the underlying machines is neither uniform nor constant over time, outliers in the assignment of items to partitions can cause data-induced load imbalance

- Micro-partitions: generate many more partitions than there are machines in the service, then do dynamic assignment and load balancing of these partitions to particular machines
- Selective replication: detect or even predict certain items that are likely to cause load imbalance and create additional replicas of these items
- Latency-induced probation

Large Information Retrieval Systems  Returning good results quickly is better than returning the best result slowly.

- Good enough: when an acceptable fraction of the overall corpus has been searched, while being careful to ensure good-enough. Good-enough schemes are also used to skip nonessential subsystems to improve responsiveness results remain rare
- Canary requests: for untested code. Rather than initially send a request to thousands of leaf servers, a root server sends it first to one or two leaf servers.

4 Other materials

4.1 Clickers

A few nodes will keep taking significantly more time than average to complete the execution of their task. This is called tail latency. This is called tail latency.

4.2 Exercise

Ne related Exercise.

4.3 Moodle

No related Moodle.

\(^3\)Applicable for reducing latency variability caused by coarser-grain phenomena
Chapter 9. Performance at Scale.

Summary of “Relational Data Processing in Spark” (Spark SQL paper)
Ivan Tishchenko, Marc Jourdan

This summary describes the mandatory Spark SQL paper. The paper is technical, low-level and contains a large degree of “noise” regarding the physical implementation of the DataFrames API and Catalyst optimizer. On the other hand, details on how to actually write queries with DataFrames were not discussed extensively in this paper and should be rather studied with the slides and a moodle quiz.

Introduction
MapReduce is very low-level and procedural way to do Big Data processing. Furthermore, it was tedious to optimize such a system for high performance. So people tried to provide a relational interface to that technology (Pig, Hive later Spark SQL etc.) to have a high level declarative language and better optimization. On the other hand, just having pure relational approach is insufficient for Big Data’s purposes. The reasons:

- users want to do ETL from various data sources (RDBMS is bad with semi-structured or unstructured data)
- users want to do complicated analytics (ML is hard to do in RDBMS)

Spark SQL integrates relational processing with Sparks functional programming API. Relational and procedural models are still disjoint and Spark SQL tries to bridge them. There is a DataFrame API which works with external data sources and also Spark’s RDDs. It has lazy evaluation for better optimizations. There is also an optimizer caller Catalyst which allows to add composable code and control code generation. Overall Spark SQL provides better optimization and richer API then bare Spark.

The paper is basically about 2 concepts. First is DataFrame which is a Dataset organized into named columns. The second one is Catalyst, which is an optimizer for DataFrames.

DataFrame:
- Offers integration of functional + relational processing styles within Spark programs
- Is a collection of structured records
- Can manipulate it with Spark or relational API
- Created from RDDs or from Java/Python objects
- Can compute multiple aggregations with just one SQL statement unlike plain RDDs
- Stored in a columnar format
- Highly optimized by Catalyst
Catalyst:
- Pattern matching to express compassable rules in a Turing complete language.
- Transforms tress
- Does logical plan analysis
- Does planning
- Performs runtime code generation.

Authors claim that Spark SQL is 10x faster than naïve Spark. Authors see it as an evolution of traditional Spark.

Motivation for Data Frames
Spark uses RDDs to build the computational DAG. RDDs are fault-tolerant and lazily evaluated to reach better optimization. Despite optimization on RDDs the engine does not understand the structure of the data inside RDDs not the semantics of user functions.

The goals of the new system are:
- Extend relational data processing to cover RDDs and other data sources.
- Gain high performance while using RDBMS techniques
- Enable the support for new data sources (semi structured data and external DBs)
- Enable extensions for Graph Processing and ML

High level overview of DataFrames API
Spark SQL runs as a library on top of Spark. The Diagram is:

![Diagram of Spark SQL and DataFrames API](image)

Main abstraction is DataFrame. It is a distributed collection of rows which have the same schema. It’s an equivalent to a table in RDBMS. But it can also be manipulated like RDDs. However unlike RDDs, DataFrames keep track of schema and support relational operations which can be optimized better than RDDs.
You can generate a DataFrame from tables in external sources, from RDDs or from Java/Python objects. You can then use standard SQL on it. DataFrames can be viewed as an RDD of Row Objects, thus you can call pure Spark’s methods such as `map` on them as well.

DataFrames are evaluated lazily i.e. DataFrame is just a logical graph on how to compute a dataset. Execution occurs only if you call an output operation (e.g. `save()`). Thus, we can do superb optimization on these DataFrames.

Spark SQL uses a nested data model based on Hive. It supports all SQL types (integer, date etc.) + complex types (struct, array, map, union). You can even nest complex types. You can also define your own types in Spark SQL.

DataFrames support all relational operators (SELECT, WHERE, GROUP BY).

To query DataFrames you can use.

1) **Relational DSL.** Code below is built as an abstract syntax tree (AST) and is passed to Catalyst for optimization later.

```scala
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

2) Use normal SQL as we did in the Moodle exercise

```scala
users.where(users("age") < 21)
  .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

You can construct a DataFrame from various data sources. Spark SQL detects names and infers types automatically. Example of generating a DataFrame from an RDD:

```scala
case class User(name: String, age: Int)

// Create an RDD of User objects
usersRDD = spark.parallelize(
  List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame
usersDF = usersRDD.toDF
```

After you’ve done such an operation Spark SQL creates a logical data scan which points to that RDD. You can even combine two different types of data sources into a single DataFrame e.g. join an RDDs and a table in Hive into one single DataFrame.

Spark SQL is able to materialize (cache) the data which is used often in columnar storage. Columnar cache significantly reduces memory requirement because it uses **dictionary encoding and run-length encoding**. You can invoke cache by calling `.cache()` on a DataFrame.
One can register User Defined Functions (UDF) by passing an inline function into the “register()” method.

**Catalyst optimizer**

The core of Spark SQL is Catalyst which does query optimization for you. Catalyst optimizer lets you easily add new optimizations techniques and is easily extendable by adding data source specific rules. Catalyst supports rule-based and cost-based optimizations.

Catalyst uses standard features of the Scala programming language like pattern matching. In its core there is a general library for representing trees and manipulating them.

Tree is the main datatype in Catalyst. A tree is the structure composed of node objects. Each node has a type and 0 or more children. An example of a tree for the expression is:

![Catalyst tree for the expression x+(1+2).](image)

We manipulate these trees using **Rules**. Rules are a function of a tree to another tree. To define such rules we used the approach of pattern matching functions that find and replace subtrees with a specific structure. Trees offer a transform method that applies pattern matching. Example of a rule definition:

```scala
    tree.transform {
        case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
    }
```

Case keyword performs pattern matching. Furthermore you can match multiple patterns in the same transform using multiple case keywords to do more complicated rules. Then you apply rules like the one above on a tree to produce another more concise tree. (e.g. \(x+(1+2)\) would be \(x+3\) after the rule).

Rules are grouped into **batches** and each batch is executed until a **fixed point** (the moment tree stops changing after the rule is applied again and again) is reached. Example: you can apply our rule many times on \((x+0)+(3+3)\) to reach a fixed point \(x+3\). Different batches generally do different tasks such as type assignment, constant folding etc. Rules can even contain Scala code. Catalyst is more powerful than DSL while it uses simple rules.
In short, Catalyst in Spark SQL is used in 4 phases

1) Analyze a logical plan
2) Optimize logical plan
3) Perform physical planning
4) Generate code and compile part of query in Java bytecode

Phases used in Catalyst:

Analysis phase
Start from AST or from DataFrame. Then the **unresolved logical plan** is built. Then we apply rules to:
- Look up the relations by name from the catalog
- Map the attributes to the input
- Give unique IDs to attributes.
- Propagate the types through expressions

Logical Optimization phase
Optimize the logical plan using rules as we discussed before. For instance fold constants, prune projections, simplify expressions etc.

Physical Planning phase
We generate multiple physical plans from logical plans. Then we select one plan using the cost model. The physical planner performs rule-based optimizations (e.g. pipeline projections into one map operation).
**Code Generation phase**
We generate actual Java bytecode to be executed on a machine. Catalyst transforms SQL to a Scala AST code for evaluation and later compilation and for running generated code. In short they use Quasiquotes from Scala programming language. Example: a tree `Add(Literal(1), Attribute("x"))` becomes an AST for a Scala expression like `1+row.get("x")`. The code additionally optimized by the Scala compiler.

Thus Catalyst is very expendable. Third parties can just add new rules and batches of rules. There are also further high level and public accessible extensions such as **data sources** and **user defined types**.

**Data Sources**
Developers will define their new DataSources. For example people implemented the interfaces for:

- CSV
- Avro
- Parquet
- JDBC

An example of using a data source.

```sql
CREATE TEMPORARY TABLE messages
USING com.databricks.spark.avro
OPTIONS (path "messages.avro")
```

**User Defined Types or UDTs**
To allow advanced analytics user can define their own types such as vector types, graphs etc.

An example of defining a type `Point(x,y)`:  
```scala
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq( // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

After registering such a type, objects of such a type will be stored in a columnar format with caching the data.
Advanced features of Catalyst
They talk about 3 advanced features they added.

First, they invented **Schema Inference for Semi structured Data** to avoid lengthy boilerplate code when parsing semi structured data such as JSON.

Second, they describe how Spark SQL is being incorporated into a new high-level API for Spark’s machine learning library.

Third feature is federation which allows a single program to efficiently query disparate sources. These features all build on the Catalyst framework.

**Schema Inference for Semi structured Data**
Does one pass over JSON and infers the typos the fields and find a corresponding Spark SQL type. The algorithm is communication efficient.

**Integration with Spark’s Machine Learning Library.**
Based on the pipelines. DataFrames is the type which exchanged between parts of the pipelines. Each column in a DataFrame will be a feature of the data. You can expose algorithms to SQL.

**Query Federation to External Databases**
Idea: avoid naïve querying since the data can be on multiple machines. Spark SQL data sources leverage Catalyst to push predicates down into the data sources whenever possible.

**Evaluation of Spark SQL**
In short, Spark SQL enables richer functionalities and brings performance increases. Also the programs are easier to understand and code according to authors. Overall, DataFrames are faster than vanilla Spark because of the Catalyst and code generation. It’s especially useful for Python apps.

![Runtime Performance Chart](image)

*Figure 10: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).*
Developers can write procedural and relational code in one program now. They claimed at least 2x performance gain when using DataFrames instead of Spark.

External research based on Catalyst
Some People used in generalized online aggregation as well as in computational genomics.

Summary
Spark SQL is library/module build on top of Spark which bridges relational and functional data processing. Spark SQL extends Spark by adding DataFrames API. DataFrames allow relational data processing, better automatic optimization, giving users the ability to write complex pipelines that mix relational and complex analytics.

Spark SQL supports large-scale data analytics, semi-structured data, query federation and custom data types for machine learning. Spark SQL uses Catalyst as an optimizer. Catalysts lets developers to add optimization rules, define data sources and new types by embedding it into the Scala programming language.