Query Optimization 2

Instructor: Matei Zaharia

cs245.stanford.edu
Outline

What can we optimize?

Rule-based optimization

Data statistics

Cost models

Cost-based plan selection

Spark SQL
Outline

What can we optimize?
Rule-based optimization
Data statistics
Cost models
Cost-based plan selection
Spark SQL
Recall From Last Time

Cost models attempt to predict a cost metric for each operator (e.g. CPU cycles, I/Os, etc)

Most common metric: # of disk I/Os
Example: Index vs Table Scan

Our query: $\sigma_p(R)$ for some predicate $p$

$s = p$’s selectivity (fraction tuples passing)

Table scan:

$R$ has $B(R) = T(R) \times S(R)/b$ blocks on disk

Cost: $B(R)$ I/Os

Index search:

Index lookup for $p$ takes $L$ I/Os

We then have to read part of $R$;
$Pr[\text{read block } i]$

$\approx 1 - Pr[\text{no match}]$

records in block

$= 1 - (1-s)^b / S(R)$

Cost: $L + (1-(1-s)^{b/S(R)}) B(R)$
What If Results Were Clustered?

We’d need to change our estimate of $C_{\text{index}}$:

$$C_{\text{index}} = L + s \ B(R)$$

Less than $C_{\text{index}}$ for unclustered data
Join Operators

Join **orders** and **algorithms** are often the choices that affect performance the most.

For a multi-way join $R \bowtie S \bowtie T \bowtie \ldots$, each join is selective and order matters a lot:

- Try to eliminate lots of records early.

Even for one join $R \bowtie S$, algorithm matters.
Example

SELECT order.date, product.price, customer.name
FROM order, product, customer
WHERE order.product_id = product.product_id
AND order.cust_id = customer.cust_id
AND product.type = "car"
AND customer.country = "US"

Plan 1:

Plan 2:

When is each plan better?
Common Join Algorithms

Iteration (nested loops) join

Merge join

Join with index

Hash join
Iteration Join

for each $r \in R_1$:
  for each $s \in R_2$:
    if $r.C == s.C$ then output $(r, s)$

I/Os: one scan of $R_1$ and $T(R_1)$ scans of $R_2$, so
cost = $B(R_1) + T(R_1) B(R_2)$ reads

Improvement: read $M$ blocks of $R_1$ in RAM at
a time then read $R_2$: $B(R_1) + B(R_1) B(R_2) / M$

Note: cost of writes is always $B(R_1 \Join R_2)$
Merge Join

if R₁ and R₂ not sorted by C then sort them
i, j = 1
while i ≤ T(R₁) && j ≤ T(R₂):
    if R₁[i].C = R₂[j].C then outputTuples
    else if R₁[i].C > R₂[j].C then j += 1
    else if R₁[i].C < R₂[j].C then i += 1
Merge Join

procedure outputTuples:
    while $R_1[i].C == R_2[j].C$ && $i \leq T(R_1)$:
        jj = j
        while $R_1[i].C == R_2[jj].C$ && $jj \leq T(R_2)$:
            output ($R_1[i]$, $R_2[jj]$)
            jj += 1
        i += i+1
## Example

<table>
<thead>
<tr>
<th>i</th>
<th>( R_1[i].C )</th>
<th>( R_2[j].C )</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>
Cost of Merge Join

If $R_1$ and $R_2$ already sorted by $C$, then

$$\text{cost} = B(R_1) + B(R_2) \text{ reads}$$

$$+ \text{ write cost of } B(R_1 \bowtie R_2)$$
Cost of Merge Join

If $R_i$ is not sorted, can sort it in $4 \cdot B(R_i)$ I/Os:

» Read runs of tuples into memory, sort
» Write each sorted run to disk
» Read from all sorted runs to merge
» Write out results
Join with Index

for each \( r \in R_1 \):
    list = index\_lookup(R_2, C, r.C)
    for each \( s \in \text{list} \):
        output \((r, s)\)

Read I/Os: 1 scan of \( R_1 \), \( T(R_1) \) index lookups on \( R_2 \), and \( T(R_1) \) data lookups

\[
\text{cost} = B(R_1) + T(R_1)(L_{\text{index}} + L_{\text{data}})
\]

Can be less when \( R_1 \) is sorted/clustered by \( C \)!
Hash Join (R₂ Fits in RAM)

hash = load R₂ into RAM and hash by C
for each r ∈ R₁:
    list = hash_lookup(hash, r.C)
    for each s ∈ list:
        output (r, s)

Read I/Os: B(R₁) + B(R₂)
Hash Join on Disk

Can be done by hashing both tables to a common set of buckets on disk

» Similar to merge sort: $4 \cdot (B(R_1) + B(R_2))$

Trick: hash only (key, pointer to record) pairs

» Can then sort the pointers to records that match and fetch them near-sequentially
Other Concerns

Join selectivity may affect how many records we need to fetch from each relation
  » If very selective, may prefer methods that join pointers or do index lookups
Summary

Join algorithms can have different performance in different situations.

In general, the following are used:

» Index join if an index exists
» Merge join if at least one table is sorted
» Hash join if both tables unsorted
Outline

What can we optimize?

Rule-based optimization

Data statistics

Cost models

Cost-based plan selection

Spark SQL
Complete CBO Process

Generate and compare possible query plans

Generate

Prune

Estimate Cost

Select

Query

Plans

Costs

Pick Min
How to Generate Plans?

Simplest way: recursive search of the options for each planning choice
How to Generate Plans?

Can limit search space: e.g. many DBMSes only consider “left-deep” joins

Often interacts well with conventions for specifying join inputs in asymmetric join algorithms (e.g. assume right argument has index)
How to Generate Plans?

Can prioritize searching through the most impactful decisions first

» E.g. join order is one of the most impactful
How to Prune Plans?

While computing the cost of a plan, throw it away if it is worse than best so far

Start with a greedy algorithm to find an “OK” initial plan that will allow lots of pruning
Memoization and Dynamic Programming

During a search through plans, many subplans will appear repeatedly.

Remember cost estimates and statistics ($T(R)$, $V(R, A)$, etc) for those: “memoization”

Can pick an order of subproblems to make it easy to reuse results (dynamic programming)
Resource Cost of CBO

It’s possible for cost-based optimization itself to take longer than running the query!

Need to design optimizer to not take too long
  » That’s why we have shortcuts in stats, etc

Luckily, a few “big” decisions drive most of the query execution time (e.g. join order)
Outline

What can we optimize?

Rule-based optimization

Data statistics

Cost models

Cost-based plan selection

Spark SQL
Background

2004: MapReduce published, enables writing large scale data apps on commodity clusters

» Cheap but unreliable “consumer” machines, so system emphasizes fault tolerance

» Focus on C++/Java programmers
Background

**2006:** Apache Hadoop project formed as an open source MapReduce + distributed FS
  » Started in Nutch open source search engine
  » Soon adopted by Yahoo & Facebook

**2006:** Amazon EC2 service launched as the newest attempt at “utility computing”
Background

2007: Facebook starts Hive (later Apache Hive) for SQL on Hadoop
  » Other SQL-on-MapReduces existed too
  » First steps toward “data lake” architecture
Background

2006-2012: Many other cluster programming frameworks proposed to bring MR’s benefits to other apps
Background

2010: Spark engine released, built around MapReduce + in-memory computing

» Motivation: interactive queries + iterative algorithms such as graph analytics

Spark then moves to be a general (“unified”) engine, covering existing ones
Code Size Comparison (2013)

- Hadoop MapReduce
- Impala (SQL)
- Storm (Streaming)
- Giraph (Graph)
- Spark

non-test, non-example source lines
Background

2012: Shark starts as a port of Hive on Spark

2014: Spark SQL starts as a SQL engine built directly on Spark (but interoperable w/ Hive)
  » Also adds two new features: DataFrames for integrating relational ops in complex programs and extensible optimizer
Original Spark API

Resilient Distributed Datasets (RDDs)
» Immutable collections of objects that can be stored in memory or disk across a cluster
» Built with parallel transformations (map, filter, …)
» Automatically rebuilt on failure
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(s => s.startswith("ERROR"))
messages = errors.map(s => s.split(\'\t\')(2))
messages.cache()

messages.filter(s => s.contains("foo")).count()
messages.filter(s => s.contains("bar")).count()
...
```

Result: full-text search of Wikipedia in 1 sec (vs 40 s for on-disk data)
Challenges with Spark’s Functional API

Looks high-level, but hides many semantics of computation from engine

» Functions passed in are arbitrary blocks of code
» Data stored is arbitrary Java/Python objects

Users can mix APIs in suboptimal ways
Example Problem

pairs = data.map(word => (word, 1))

groups = pairs.groupByKey()

groups.map((k, vs) => (k, vs.sum))

Materializes all groups as lists of integers

Then promptly aggregates them
Challenge: Data Representation

Java objects often many times larger than data

class User(name: String, friends: Array[Int])
User(“Bobby”, Array(1, 2))
Spark SQL & DataFrames

Efficient library for working with structured data
» 2 interfaces: SQL for data analysts and external apps, DataFrames for complex programs
» Optimized computation and storage underneath
Spark SQL Architecture

- Logical Plan
- Physical Plan
- Optimizer
- Code Generator
- RDDs
- Catalog
- Data Source API (HDFS, Cassandra, HBase, Elasticsearch, PostgreSQL, Hive, ...)
DataFrame API

DataFrames hold rows with a known schema and offer relational operations through a DSL.

c = HiveContext()
users = c.sql("select * from users")

ma_users = users[users.state == "MA"]

ma_users.count()

ma_users.groupBy("name").avg("age")

ma_users.map(lambda row: row.user.toUpper())
API Details

Based on data frame concept in R, Python
» Spark is the first to make this declarative

Integrated with the rest of Spark
» ML library takes DataFrames as input/output
» Easily convert RDDs ↔ DataFrames

Google trends for “data frame”
What DataFrames Enable

1. Compact binary representation
   • Columnar, compressed cache; rows for processing

2. Optimization across operators (join reordering, predicate pushdown, etc)

3. Runtime code generation
Performance

Time for aggregation benchmark (s)

- RDD Python
- RDD Scala
Performance

- DataFrame SQL
- DataFrame R
- DataFrame Python
- DataFrame Scala
- RDD Python
- RDD Scala

Time for aggregation benchmark (s)
Data Sources

Uniform way to access structured data
  » Apps can migrate across Hive, Cassandra, JSON, Parquet, …
  » Rich semantics allows query pushdown into data sources

users[users.age > 20]
select * from users
Examples

**JSON:**

```json
{
  "text": "hi",
  "user": {
    "name": "bob",
    "id": 15
  }
}
```

**JDBC:**

```sql
select age from users where lang = "en"
```

**Together:**

```sql
select t.text, u.age
from tweets t, users u
where t.user.id = u.id
and u.lang = "en"
```

**Spark SQL**

```sql
select id, age from users where lang="en"
```
Extensible Optimizer

Uses Scala pattern matching (see demo!)
Which Spark Components Do People Use?

- Spark SQL: 69%
- DataFrames: 62%
- Spark Streaming: 58%
- MLlib + GraphX: 58%

75% of users use 2 or more components

(2015 survey)
Which Languages Are Used?

2014 Languages Used
- Scala: 84%
- Java: 38%
- Python: 38%

2015 Languages Used
- Scala: 71%
- Java: 31%
- Python: 58%
- R: 18%
Extensions to Spark SQL

Tens of data sources using the pushdown API

Interval queries on genomic data

Geospatial package (Magellan)

Approximate queries & other research