Query Optimization 2

Instructor: Matei Zaharia

cs245.stanford.edu
Recap: Data Statistics

Information about tuples in a table that we can use to estimate costs

» Must be *approximated* for intermediate tables

We saw one way to do this for 4 statistics:

» \( T(R) = \# \) of tuples in \( R \)
» \( S(R) = \) average size of tuples in \( R \)
» \( B(R) = \# \) of blocks to hold \( R \)’s tuples
» \( V(R, A) = \# \) distinct values of attribute \( A \) in \( R \)
Another Type of Data Stats: Histograms

The diagram shows a histogram where:

- The x-axis represents ranges of numbers from 10 to 40.
- The y-axis represents the number of tuples in R with A values in a given range.
- The bars indicate the count of tuples for each range:
  - 10: 5
  - 20: 10
  - 30: 15
  - 40: 12

The expression 
\[ \sigma_{A \geq a}(R) = ? \]
represents the number of tuples in R with A values greater than or equal to a.
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
Cost Models

How do we measure a query plan’s cost?

Many possible metrics:

» Number of disk I/Os
» Number of compute cycles
» Combined time metric
» Memory usage
» Bytes sent on network
» …
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 $R$;
  - $Pr[read \ block \ i] \\ = 1 - Pr[no \ match]^{records \ in \ block}$
  - $= 1 - (1-s)^b/S(R)$
- Cost: $L + (1-(1-s)^b/S(R)) \times B(R)$
What If Results Were Clustered?

Unclustered: records that match p are spread out uniformly

Clustered: records that match p are close together in R’s file

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

\[
C_{\text{index}} = L + s \cdot B(R)
\]

Less than \( C_{\text{index}} \) for unclustered data

Fraction of R’s blocks read
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:

order
 product (type=car)

customer (country=US)

Plan 2:

order

customer (country=US)

product (type=car)

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
\[
\text{cost} = B(R_1) + T(R_1) B(R_2)
\]

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 \( B(R_1 \bowtie R_2) \)
Merge Join

if $R_1$ and $R_2$ not sorted by $C$ then sort them
$i, j = 1$
while $i \leq T(R_1)$ && $j \leq T(R_2)$:
    if $R_1[i].C = R_2[j].C$ then outputTuples
    else if $R_1[i].C > R_2[j].C$ then $j += 1$
    else if $R_1[i].C < R_2[j].C$ 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></td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
<td>7</td>
</tr>
</tbody>
</table>
Cost of Merge Join

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

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

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

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 \( r \in R₁: \)
    list = hash_lookup(hash, r.C)
    for each \( s \in list: \)
        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: \(4 \times (B(R_1) + B(R_2))\)

**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 \((T(R), V(R, A), \text{etc})\) 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 models to bring MR’s benefits to other apps

- Pregel
- Dremel
- Impala
- Dryad
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)

non-test, non-example source lines
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 DataFrames for integrating relational ops in Scala/Java/Python programs
Original Spark API

Resilient Distributed Datasets (RDDs)

- Immutable collections of objects that can be stored in memory or disk across a cluster
- Built via 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

```scala
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()
...
```

Interactive ad-hoc queries in your favorite language
Challenges with Spark’s Functional API

Looks high-level, but hides many semantics of computation from engine
  » Functions passed in are arbitrary 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))
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 & storage underneath
Spark SQL Architecture

SQL → Logical Plan

Data Frames → Logical Plan

Optimizer → Physical Plan

Physical Plan → Code Generator → RDDs

Catalog → Data Source API

Data Source API supports:
- 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()  

maUsers.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:

select user.id, text from tweets

JDBC:

select age from users where lang = "en"

Together:

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

Uses Scala pattern matching (see demo!)
Spark Usage Today

Languages Used in Databricks Notebooks

- Python: 68%
- SQL: 18%
- Scala: 11%
- R: 3%

>90% of API calls run via Spark SQL engine
Extensions to Spark SQL

Structured Streaming (streaming SQL)

Many data sources using the pushdown API

Interval queries on genomic data

Geospatial package (Magellan)