Query Execution

Instructor: Matei Zaharia

cs245.stanford.edu
From Last Time: Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing
Example

Find records where

DEPT = “Toy” AND SALARY > 50k
Strategy I:

Use one index, say Dept.

Get all Dept = “Toy” records and check their salary
Strategy II:

Use 2 indexes; manipulate record pointers

Toy $\rightarrow$ \[\begin{array}{c}
\end{array}\] \[\begin{array}{c}
\end{array}\] $\leftarrow$ Sal

\[\text{> 50k}\]
Strategy III: Multi-key index

One idea:
Example

Example Record

Name=Joe
DEPT=Sales
SALARY=15k
k-d Tree

Splits dimensions in any order to hold k-dimensional data
k-d Tree
k-d Tree

Diagram of a k-d Tree with nodes and coordinates.
k-d Tree
k-d Tree

Efficient range queries in both dimensions
Summary

Wide range of indexes for different data types and queries (e.g. range vs exact)

Issues to balance: query time, update cost, and size of index
Example Storage Strategies

MySQL: transactional DBMS
  » Row-oriented storage with 16 KB pages
  » Variable length records with headers, overflow
  » Index types:
    • B-tree
    • Hash (in memory only)
    • R-tree (spatial data)
    • Inverted lists for full text search
  » Can compress pages with Lempel-Ziv
Example Storage Strategies

Apache Parquet + Hive: analytical data lake
  » Column-oriented storage as set of ~1 GB files (each file has a slice of all columns)
  » Various compression and encoding schemes at the level of pages in a file
    • Special scheme for nested fields (Dremel)
  » Header with statistics at the start of each file
    • Min/max of columns, nulls, Bloom filter
  » Files partitioned into directories by one key
Query Execution

Overview

Relational operators

Execution methods
Query Execution Overview

Recall that one of our key principles in data intensive systems was **declarative APIs**

» Specify what you want to compute, not how

We saw how these can translate into many storage strategies

How to execute queries in a declarative API?
Query Execution Overview

- Query representation (e.g. SQL)
- Logical query plan (e.g. relational algebra)
- Optimized logical plan
- Physical plan (code/operators to run)

Many execution methods: per-record exec, vectorization, compilation
Plan Optimization Methods

**Rule-based**: systematically replace some expressions with other expressions
  » Replace X OR TRUE with TRUE
  » Replace M*A + M*B with M*(A+B) for matrices

**Cost-based**: propose several execution plans and pick best based on a cost model

**Adaptive**: update execution plan at runtime
Execution Methods

**Interpretation:** walk through query plan operators for each record

**Vectorization:** walk through in batches

**Compilation:** generate code (like System R)
Typical RDBMS Execution

- SQL query
  - parse
    - parse tree
  - convert
    - logical query plan
  - apply rules
    - “improved” l.q.p
  - estimate result sizes
    - l.q.p. + sizes
  - consider physical plans
- {P₁, P₂, …}

- estimate costs
- pick best
  - {P₁, C₁), (P₂, C₂), …}
- execute
- result
Query Execution

Overview

Relational operators

Execution methods
The Relational Algebra

Collection of operators over tables (relations)
» Each table has named attributes (fields)

Codd’s original RA: tables are sets of tuples (unordered and tuples cannot repeat)

SQL’s RA: tables are bags (multisets) of tuples; unordered but each tuple may repeat
Relational Algebra Operators

Basic set operators:

**Intersection:** $R \cap S$

**Union:** $R \cup S$

**Difference:** $R - S$

**Cartesian Product:** $R \times S = \{(r, s) \mid r \in R, s \in S\}$ for tables with same schema
Relational Algebra Operators

Basic set operators:

**Intersection:** \( R \cap S \)

**Union:** \( R \cup S \) \( \quad \) consider both distinct (set union) and non-distinct (bag union)

**Difference:** \( R - S \)

**Cartesian Product:** \( R \times S \)
Relational Algebra Operators

Special query processing operators:

**Selection:** $\sigma_{\text{condition}}(R)$ \{ $r \in R$ | condition($r$) is true \}

**Projection:** $\Pi_{\text{expressions}}(R)$ \{ expressions($r$) | $r \in R$ \}

**Natural Join:** $R \bowtie S$ \{ $(r, s) \in R \times S$ | $r$.key = $s$.key \}

where key is the common fields
Relational Algebra Operators

Special query processing operators:

**Aggregation:** \( \text{keys} \ G_{\text{agg(attr)}}(R) \)  \[ \text{SELECT} \ \text{agg(attr)} \ \text{FROM} \ R \ \text{GROUP BY} \ \text{keys} \]

**Examples:**

- \( \text{department} \ G_{\text{Max(salary)}}(\text{Employees}) \)
- \( G_{\text{Max(salary)}}(\text{Employees}) \)
Algebraic Properties

Many properties about which combinations of operators are equivalent
  » That’s why it’s called an algebra!
Properties: Unions, Products and Joins

\[ R \cup S = S \cup R \]
\[ R \cup (S \cup T) = (R \cup S) \cup T \]

\[ R \times S = S \times R \]
\[ (R \times S) \times T = R \times (S \times T) \]

\[ R \bowtie S = S \bowtie R \]
\[ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \]

Tuple order in a relation doesn’t matter (unordered)

Attribute order in a relation doesn’t matter either
Properties: Selects

\[ \sigma_{p \land q}(R) = \]

\[ \sigma_{p \lor q}(R) = \]
Properties: Selects

\[ \sigma_{p \land q}(R) = \sigma_p(\sigma_q(R)) \]

\[ \sigma_{p \lor q}(R) = \sigma_p(R) \cup \sigma_q(R) \]

careful with repeated elements
Bags vs. Sets

R = \{a,a,b,b,b,c\}

S = \{b,b,c,c,d\}

R \cup S = ?
Bags vs. Sets

\[ R = \{a,a,b,b,b,c\} \]
\[ S = \{b,b,c,c,d\} \]
\[ R \cup S = ? \]

- **Option 1:** SUM of counts
  \[ R \cup S = \{a,a,b,b,b,b,b,c,c,c,d\} \]

- **Option 2:** MAX of counts
  \[ R \cup S = \{a,a,b,b,b,c,c,d\} \]
Executive Decision

Use “SUM” option for bag unions

Some rules that work for set unions cannot be used for bags
Properties: Project

Let: \( X = \) set of attributes
    \( Y = \) set of attributes

\( \Pi_{X \cup Y} (R) = \)
Properties: Project

Let:  \( X = \text{set of attributes} \)

\( Y = \text{set of attributes} \)

\( \Pi_{X \cup Y}(R) = \Pi_X(\Pi_Y(R)) \)
Properties: Project

Let: \( X \) = set of attributes
     \( Y \) = set of attributes

\[ \Pi_{X \cup Y}(R) = \Pi_X(\Pi_Y(R)) \]
Properties: $\sigma + \Join$

Let $p = \text{predicate with only R attribs}$

$q = \text{predicate with only S attribs}$

$m = \text{predicate with only R, S attribs}$

$\sigma_p(R \Join S) =$

$\sigma_q(R \Join S) =$
Properties: $\sigma + \Join$

Let $p =$ predicate with only $R$ attribs

$q =$ predicate with only $S$ attribs

$m =$ predicate with only $R$, $S$ attribs

\[
\sigma_p(R \Join S) = \sigma_p(R) \Join S
\]

\[
\sigma_q(R \Join S) = R \Join \sigma_q(S)
\]
Properties: $\sigma + \Join$

Some rules can be derived:

$\sigma_{p \land q}(R \Join S) =$

$\sigma_{p \land q \land m}(R \Join S) =$

$\sigma_{p \lor q}(R \Join S) =$
Properties: $\sigma + \Join$

Some rules can be derived:

$$\sigma_{p \land q}(R \Join S) = \sigma_p(R) \Join \sigma_q(S)$$

$$\sigma_{p \land q \land m}(R \Join S) = \sigma_m(\sigma_p(R) \Join \sigma_q(S))$$

$$\sigma_{p \lor q}(R \Join S) = (\sigma_p(R) \Join S) \cup (R \Join \sigma_q(S))$$
Prove One, Others for Practice

\[ \sigma_{p \land q}(R \bowtie S) = \sigma_p (\sigma_q(R \bowtie S)) \]

\[ = \sigma_p (R \bowtie \sigma_q(S)) \]

\[ = \sigma_p (R) \bowtie \sigma_q(S) \]
Properties: $\Pi + \sigma$

Let $x$ = subset of $R$ attributes

$z$ = attributes in predicate $p$
  (subset of $R$ attributes)

$\Pi_x(\sigma_p (R)) =$
Properties: $\Pi + \sigma$

Let $x$ = subset of $R$ attributes

$z = \text{attributes in predicate } p$

(subset of $R$ attributes)

$\Pi_x(\sigma_p(R)) = \sigma_p(\Pi_x(R))$
Properties: $\Pi + \sigma$

Let $x = \text{subset of } R \text{ attributes}$

\[
z = \text{attributes in predicate } p
\]
\[
\text{(subset of } R \text{ attributes)}
\]

\[
\Pi_x(\sigma_{p(R)}) = \Pi_x(\sigma_{p(\Pi_{x\cup z}(R))})
\]
Properties: $\Pi + \Join$

Let $x$ = subset of $R$ attributes

$y$ = subset of $S$ attributes

$z$ = intersection of $R, S$ attributes

$\Pi_{x\cup y}(R \Join S) = \Pi_{x\cup y}((\Pi_{x\cup z}(R)) \Join (\Pi_{y\cup z}(S)))$
Typical RDBMS Execution

1. SQL query
2. Parse
   - Parse tree
3. Convert
   - Logical query plan
4. Apply rules
   - "Improved" l.q.p.
5. Estimate result sizes
   - L.q.p. + sizes
6. Consider physical plans
7. Estimate costs
   - Statistics
   - \( \{ (P_1, C_1), (P_2, C_2), \ldots \} \)
8. Pick best
   - \( P_i \)
9. Execute
10. Result

\[ \{ P_1, P_2, \ldots \} \]
Example SQL Query

SELECT title
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE ‘%1960’
);

(Find the movies with stars born in 1960)
Parse Tree

SELECT <SelList> FROM <FromList> WHERE <Condition>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>RelName</th>
<th>Tuple</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>StarsIn</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>starName</td>
<td></td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>MovieStar</td>
<td>LIKE</td>
<td>‘%1960’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logical Query Plan

\[
\begin{align*}
\Pi_{\text{title}} \\
\sigma_{\text{starName}=\text{name}} \\
\times \\
\Pi_{\text{name}} \\
\sigma_{\text{birthdate LIKE} \, \%1960} \\
\times \\
\text{MovieStar}
\end{align*}
\]
**Improved Logical Query Plan**

\[ \Pi_{\text{title}} \]

\[ \starName = \text{name} \]

\[ \Pi_{\text{name}} \]

\[ \sigma \text{birthdate LIKE '1960'} \]

\[ \text{MovieStar} \]

**Question:**
Push \( \Pi_{\text{title}} \) to \( \text{StarsIn} \)?
Estimate Result Sizes

\[ \Pi \sigma \]

StarsIn

Need expected size

MovieStar
One Physical Plan

Hash join

Seq scan
StarsIn

Index scan
MovieStar

Parameters: join order, memory size, project attributes, ...

Parameters: select condition, ...
Another Physical Plan

Hash join

Index scan
StarsIn

Seq scan
MovieStar

Parameters: join order, memory size, project attributes, ...

Parameters: select condition, ...
Another Physical Plan

Which plan is likely to be better?
Estimating Plan Costs

Logical plan

\[
P_1 \quad P_2 \quad \ldots \quad P_n
\]

\[
C_1 \quad C_2 \quad \ldots \quad C_n
\]

Pick best!

Covered in next few lectures!
Query Execution

Overview

Relational operators

Execution methods
Now That We Have a Plan, How Do We Run it?

Several different options that trade between complexity, setup time & performance
Example: Simple Query

\[ \Pi_{\text{quantity} \times \text{price}} (\sigma_{\text{productId} = 75} (\text{orders})) \]
Method 1: Interpretation

interface Operator {
    Tuple next();
}

class TableScan: Operator {
    String tableName;
}

class Select: Operator {
    Operator parent;
    Expression condition;
}

class Project: Operator {
    Operator parent;
    Expression[] exprs;
}

interface Expression {
    Value compute(Tuple in);
}

class Attribute: Expression {
    String name;
}

class Times: Expression {
    Expression left, right;
}

class Equals: Expression {
    Expression left, right;
}
Example Expression Classes

class Attribute: Expression {
    String name;

    Value compute(Tuple in) {
        return in.getField(name);
    }
}

class Times: Expression {
    Expression left, right;

    Value compute(Tuple in) {
        return left.compute(in) * right.compute(in);
    }
}
Example Operator Classes

class TableScan: Operator {
    String tableName;

    Tuple next() {
        // read next record from file
    }
}

class Project: Expression {
    Operator parent;
    Expression[] exprs;

    Tuple next() {
        tuple = parent.next();
        fields = [expr(tuple) for expr in exprs];
        return new Tuple(fields);
    }
}
Running Our Query with Interpretation

ops = Project(
    expr = Times(Attr("quantity"), Attr("price")),
    parent = Select(
        expr = Equals(Attr("productId"), Literal(75)),
        parent = TableScan("orders")
    )
);

while(true) {
    Tuple t = ops.next();
    if (t != null) {
        out.write(t);
    } else {
        break;
    }
}
Method 2: Vectorization

Interpreting query plans one record at a time is simple, but it’s too slow

» Lots of virtual function calls and branches for each record (recall Jeff Dean’s numbers)

Keep recursive interpretation, but make Operators and Expressions run on batches
Implementing Vectorization

class TupleBatch {
    // Efficient storage, e.g.
    // schema + column arrays
}

interface Operator {
    TupleBatch next();
}

class Select: Operator {
    Operator parent;
    Expression condition;
}

...
Typical Implementation

Values stored in columnar arrays (e.g. int[]) with a separate bit array to mark nulls

Tuple batches fit in L1 or L2 cache

Operators use SIMD instructions to update both values and null fields without branching
Pros & Cons of Vectorization

+ Faster than record-at-a-time if the query processes many records
+ Relatively simple to implement
  – Lots of nulls in batches if query is selective
  – Data travels between CPU & cache a lot
Method 3: Compilation

Turn the query into executable code
Compilation Example

\[ \Pi_{\text{quantity*price}} (\sigma_{\text{productId}=75} (\text{orders})) \]

class MyQuery {
    void run() {
        Iterator<OrdersTuple> in = openTable("orders");
        for(OrdersTuple t: in) {
            if (t.productId == 75) {
                out.write(Tuple(t.quantity * t.price));
            }
        }
    }
}

generated class with the right field types for orders table

Can also theoretically generate vectorized code
Pros & Cons of Compilation

+ Potential to get fastest possible execution
+ Leverage existing work in compilers
  – Complex to implement
  – Compilation takes time
  – Generated code may not match hand-written
What’s Used Today?

Depends on context & other bottlenecks

Transactional databases (e.g. MySQL): mostly record-at-a-time interpretation

Analytical systems (Vertica, Spark SQL): vectorization, sometimes compilation

ML libs (TensorFlow): mostly vectorization (the records are vectors!), some compilation