Databases and information management

Lecture 14

Piotr Syga

2023-01-19

The optimizer

1. Verification of the syntax
2. Identification of the required nodes of distributed database
3. Identification of the required tables and views
4. Access level control
5. Checking the existence and applicability of the indexes
6. Simplification of the query
7. Setting the priorities to the subqueries in order to reduce the delay

Phases of query processing

1. Decomposition of complex queries (including distributed ones)
2. Transcription to Relation Algebra
3. Creating a query graph (tree preferably)
4. Query consistency and validity check
5. Simplification of the query
6. Generation of the code
7. Returning the result

Types of optimization

- Optimization w.r.t. the number of operations vs w.r.t. the cost
- Sequential vs batch
- Static vs dynamic

Static optimization

- optimizing a SQL query before it is executed
- analyzing the query's syntax and semantics
- structure of the underlying data and statistics of presence of particular data may be taken into account but is not required
- the goal is to determine the most efficient execution plan based only on the information available at the time the query is compiled

Dynamic optimization

- optimizing a SQL during execution
- monitoring the query's performance as it runs
- real-time adjustments to the execution plan
- the goal is to adapt to the changing conditions (network, CPU usage, etc.) and ensure the best possible performance

Decomposition stages

- Syntax verification
- Checking the availability of the required relations and attributes
- Transcribing to RA
- Creation of query graph
- Specifying the contexts

```
SELECT * FROM Students JOIN Courses ON
Student.index=Courses.student WHERE Student.name='Joe' AND
Courses.day='Thursday';
```

Normalization of the query, simplification of the conditions

- *Fact*: each 1st order logic sentence might be rewritten using only ($\neg$, $\wedge$, $\vee$)
- Conjunctive normal form vs Disjunctive normal form
- detection of erros or contradictory conditions
- construction of relation cyclic graph (RCG) – consistency verification
- removing redundant requirements, tautologies, antitautologies, ...
- the tree is build from the leaves (relations) up to the root (`SELECT`)
- optionally subtrees may be labeled for parallel processing

Relation cyclic graph

- a graph representation of relationships between the tables in the database
- each node represents a table (or a view) and each edge – a connection
- the edges are directed and may be labeled(e.g., one-to-one, one-to-many, many-to-many)
- RCG can detect missing data sources in the query (when the graph is disjoint)
- RCG can be used to detect cycles in the relationships between tables (when query is trying to join a table to itself through a chain of relationships)

Atribute Connection Graph

- each node represents an attribute in the database
- each edge represents a relationship between two attributes
- the edges are directed
- ACG can be used to find contradictory conditions
- ACG may help us to find tautologies

Atribute Connection Graph

Analysis of ACG

Each reference to an attribute a is represented by a copy of a vertex a, in addition we create a root vertex $\mathcal{S}$. For each selected vertex a, we create an edge $a \rightarrow \mathcal{S}$ with weight 0. Furthermore, we create an edge $a \rightarrow a_j$ with weight w for the conditions $a \leq a_j + w$ (consequently, for $=$, we have two edges with weight 0), $\mathcal{S} \rightarrow_w a$ $(a \geq w)$.
We reject ACGs containing cycles with negative weights as contradictory.

ACG examples

```
SELECT D.no, D.name, D.head FROM Students S JOIN Courses C
ON S.index=C.student JOIN Department D WHERE
YEAR(S.DoB)<1990 AND D.size>5000;
```

ACG examples

```
SELECT D.no, D.name, D.head FROM Students S JOIN Courses C
ON S.index=C.student JOIN Department D WHERE
YEAR(S.DoB)<1990 AND D.size>5000;
```

Connection lacking between `D`, `S` and `C` – the graph is disjoint

ACG examples

```
SELECT D.no, D.name, D.head FROM Students S JOIN Courses C
ON S.index=C.student JOIN Department D WHERE
YEAR(S.DoB)<1990 AND D.size>5000;
```

Connection lacking between D, S and C – the graph is disjoint

```
SELECT * FROM Students S JOIN Courses C ON
S.index=C.student JOIN Department D ON C.dept=D.no WHERE
YEAR(S.DoB)<1990 AND D.size>5000 AND YEAR(S.DoB)>1999;
```

ACG examples

```
SELECT D.no, D.name, D.head FROM Students S JOIN Courses C
ON S.index=C.student JOIN Department D WHERE
YEAR(S.DoB)<1990 AND D.size>5000;
```

Connection lacking between D, S and C – the graph is disjoint

```
SELECT * FROM Students S JOIN Courses C ON
S.index=C.student JOIN Department D ON C.dept=D.no WHERE
YEAR(S.DoB)<1990 AND D.size>5000 AND YEAR(S.DoB)>1999;
```

Negative total weight cycle – YEAR(S.DoB) and $\mathcal{S}$.

```
SELECT * FROM Students S JOIN Courses C ON
S.index=C.student JOIN Department D ON C.dept=D.no WHERE
S.DoB<'1990-01-01' AND D.size>5000 AND YEAR(S.DoB)>1999;
```

What do we do in the above case of type mismatch? How do we cope with `Student.name='Joe'` condition?

Heuristics for the tree construction

- unary operators are moved towards the leaves, binary towards the root
- we use distributive and commutative properties of the binary operators that are closer to the leaves
- grouping and aggregation is moved towards the leaves
- dimension reducing operations (e.g., σ, π) or moved towards the leaves if that doesn't change the scope of the query
- binary operations may be simplified using selection ($\sigma_C(R_1 \bowtie R_2)$, $\sigma_C(R_1 \cup R_2)$, $\sigma_C(R_1 \setminus R_2)$, $\sigma_{C_1 \wedge C_2}(R)$)
- subqueries and nested queries require a separate (sub)tree, we try unnesting if possible

Unnesting

- 'flattening' the nested query into the parent query
- may be done by replacing the query with the result, utilizing the subquery into the outer query's join or filter conditions
- reduces the number of times the database needs to access the same data
- further amplified with indexes
- may increase the complexity of the query and make it harder for the optimizer to find the most efficient execution plan

Moving the filtering towards the leaves

```
SELECT C.dept, MIN(C.group_no) AS grp FROM Courses C GROUP
BY C.dept HAVING grp<10;

SELECT C.dept, MIN(C.group_no) AS grp FROM Courses C WHERE
C.group_no<10 GROUP BY C.dept;

SELECT C.dept, MIN(C.group_no) AS grp FROM Courses C GROUP
BY C.dept;
```

Bounds on the cardinality

S: `SELECT * FROM R1, R2, R3 WHERE C1 AND C2`

The upper bound

The upper bound on the number of resulting tuples is not greater than $r_1 \times r_2 \times r_3$ for r_i being the number of tuples in `Ri`.

Selection factor

Each condition `C` has a selection factor $\mathrm{sf}(\mathtt{C})$ associated with it that describes the expected value of tuples reduction

$$\mathrm{card}(S) = (r_1 \times r_2 \times r_3) \cdot \mathrm{sf}(C_1) \cdot \mathrm{sf}(C_2)$$

Estimating the projection size

P: `SELECT DISTINCT a FROM R;`

$\mathrm{card}(P) = \mathrm{val}(a[R])$, where $\mathrm{val}(\cdot)$ denotes the number of unique values of `a` in `R`.

Size

The size of the result of a query ($\mathrm{size}(P)$) is equal to the sum of sizes of all the attributes that we project on.

Gruping

Let G denote a set of attributes on which we are grouping in query S on relation R. Then $\mathrm{card}(S) \leq \Pi_{a \in G}\ \mathrm{val}(a[R])$.
The number of values: $\mathrm{val}(a[S]) = \mathrm{val}(a[R])$.
The size of the result is equal to th size of projecting R on G.

Union and Join

U: R $\cup$ S and J: R $\bowtie$ S

Cardinality : $\mathrm{card}(U) \leq \mathrm{card}(R) + \mathrm{card}(S)$,
$\mathrm{card}(J) \leq \mathrm{card}(R) \cdot \mathrm{card}(S)$ (if we assume that we Join on a: $\mathrm{card}(J) \leq \frac{\mathrm{card}(R) \cdot \mathrm{card}(S)}{\max(\mathrm{val}(a[R]), \mathrm{val}(a[S]))}$)

Size : The size of U, R and S has to be equal in case of union.
For JOIN on a group of attributes g:
$\mathrm{size}(J) = \mathrm{size}(R) + \mathrm{size}(S) - \mathrm{size}(g)$

Number of values : $\mathrm{val}(a[U]) \leq \mathrm{val}(a[R]) + \mathrm{val}(a[S])$ and
$\mathrm{val}(a[J]) \leq \mathrm{val}(a[R]) + \mathrm{val}(a[S])$ if a is not the joining attribute and $\mathrm{val}(a[J]) \leq \min(\mathrm{val}(a[R]), \mathrm{val}(a[S]))$ otherwise

Selecting the type of the tree

- left deep
- right deep
- bushy tree (Snowstorm - description, patent)

Selection of the optimal tree is feasible only for small trees, hence for more complicated queries we do use limited iterated search, heuristics and other techniques.