Database Management Systems

Query optimization
Query optimizer

- It selects an efficient strategy for query execution
  - It is a fundamental building block of a relational DBMS

- It guarantees the *data independence* property
  - The form in which the SQL query is written does not affect the way in which it is implemented
  - A physical reorganization of data does not require rewriting SQL queries
Query optimizer

- It automatically generates a *query execution plan*
  - It was formerly hard-coded by a programmer
- The automatically generated execution plan is usually more efficient
  - It evaluates many different alternatives
  - It exploits statistics on data, stored in the system catalog, to make decisions
  - It exploits the best known strategies
  - It dynamically adapts to changes in the data distribution
Query optimizer

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS
Lexical, syntactic and semantic analysis

Analysis of a statement to detect

- Lexical errors
  - e.g., misspelled keywords
- Syntactic errors
  - errors in the grammar of the SQL language
- Semantic errors
  - references to objects which do not actually exist in the database (e.g., attributes or tables)
  - information in the data dictionary is needed
Lexical, syntactic and semantic analysis

Output

- Internal representation in (extended) relational algebra

Why relational algebra?

- It explicitly represents the order in which operators are applied
  - It is *procedural* (different from SQL)
- There is a corpus of theorems and properties
  - Exploited to modify the initial query tree
Query optimizer

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

DATA DICTIONARY
Algebraic optimization

- Execution of algebraic transformations considered to be always beneficial
  - Example: anticipation of selection with respect to join
- Should eliminate the difference among different formulations of the same query
- This step is usually independent of the data distribution
- Output
  - Query tree in “canonical” form
Query optimizer

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

"CANONICAL" QUERY TREE

COST BASED OPTIMIZATION

DATA DICTIONARY
Cost based optimization

- Selection of the “best” execution plan by evaluating *execution cost*
  - Selection of
    - the best access method for each table
    - the best algorithm for each relational operator among available alternatives
  - Based on a cost model for access methods and algorithms
- Generation of the code implementing the best strategy
Cost based optimization

Output

- Access program in executable format
  - It exploits the internal structures of the DBMS
- Set of dependencies
  - conditions on which the validity of the query plan depends
    - e.g., the existence of an index
Query optimizer

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

“CANONICAL” QUERY TREE

COST BASED OPTIMIZATION

DATA DICTIONARY

DATA PROFILES (STATISTICS ON DATA)

ACCESS PROGRAM

SET OF DEPENDENCIES
Compile and go

- Compilation and *immediate* execution of the statement
- No storage of the query plan
- Dependencies are not needed
Compile and store

- The access plan is stored in the database together with its dependencies
- It is executed *on demand*
- It should be recompiled when the data structure changes
Database Management Systems

Algebraic optimization
Algebraic optimization

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

“CANONICAL” QUERY TREE

COST BASED OPTIMIZATION

DATA DICTIONARY

DATA PROFILES (STATISTICS ON DATA)

ACCESS PROGRAM

SET OF DEPENDENCIES
Algebraic optimization

▷ It is based on equivalence transformations
  • Two relational expressions are *equivalent* if they both produce the same query result for any arbitrary database instance

▷ Interesting transformations
  • reduce the size of the intermediate result to be stored in memory
  • prepare an expression for the application of a transformation which reduces the size of the intermediate result
1. Atomization of selection

\[ \sigma_{F_1 \land F_2} (E) \equiv \sigma_{F_2} (\sigma_{F_1} (E)) \equiv \sigma_{F_1} (\sigma_{F_2} (E)) \]
Transformations

1. Atomization of selection
   \[ \sigma_{F_1 \land F_2}(E) \equiv \sigma_{F_2}(\sigma_{F_1}(E)) \equiv \sigma_{F_1}(\sigma_{F_2}(E)) \]

2. Cascading projections
   \[ \pi_X(E) \equiv \pi_X(\pi_{X,Y}(E)) \]
1. Atomization of selection
   - $\sigma_{F_1 \land F_2}(E) \equiv \sigma_{F_2}(\sigma_{F_1}(E)) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$

2. Cascading projections
   - $\pi_X(E) \equiv \pi_X(\pi_{X,Y}(E))$

3. Anticipation of selection with respect to join (pushing selection down)
   - $\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie (\sigma_F(E_2))$
   - F is a predicate on attributes in $E_2$ only
4. Anticipation of projection with respect to join

\[ \pi_L(E_1 \bowtie_p E_2) \equiv \pi_L((\pi_{L1}, J(E_1)) \bowtie_p (\pi_{L2}, J(E_2))) \]

- \( L1 = L - \text{Schema}(E_2) \)
- \( L2 = L - \text{Schema}(E_1) \)
- \( J = \text{set of attributes needed to evaluate join predicate } p \)
5. Join derivation from Cartesian product

- \( \sigma_F (E_1 \times E_2) \equiv E_1 \bowtie_F E_2 \)
- predicate F only relates attributes in \( E_1 \) and \( E_2 \)
5. Join derivation from Cartesian product

- $\sigma_F (E_1 \times E_2) \equiv E_1 \bowtie_F E_2$
- predicate $F$ only relates attributes in $E_1$ and $E_2$

6. Distribution of selection with respect to union

- $\sigma_F (E_1 \cup E_2) \equiv (\sigma_F (E_1)) \cup (\sigma_F (E_2))$
5. Join derivation from Cartesian product
   - $\sigma_F(E_1 \times E_2) \equiv E_1 \bowtie_F E_2$
   - predicate $F$ only relates attributes in $E_1$ and $E_2$

6. Distribution of selection with respect to union
   - $\sigma_F(E_1 \cup E_2) \equiv (\sigma_F(E_1)) \cup (\sigma_F(E_2))$

7. Distribution of selection with respect to difference
   - $\sigma_F(E_1 - E_2) \equiv (\sigma_F(E_1)) - (\sigma_F(E_2))$
   - $\equiv (\sigma_F(E_1)) - E_2$
8. Distribution of projection with respect to union

\[ \pi_X(E_1 \cup E_2) \equiv (\pi_X(E_1)) \cup (\pi_X(E_2)) \]
8. Distribution of projection with respect to union

\[ \pi_X(E_1 \cup E_2) \equiv (\pi_X(E_1)) \cup (\pi_X(E_2)) \]

▶ Can projection be distributed with respect to difference?

\[ \pi_X(E_1 - E_2) \equiv (\pi_X(E_1)) - (\pi_X(E_2)) \]
8. Distribution of projection with respect to union

- $\pi_X(E_1 \cup E_2) \equiv (\pi_X(E_1)) \cup (\pi_X(E_2))$

Can projection be distributed with respect to difference?

- $\pi_X(E_1 - E_2) \not\equiv (\pi_X(E_1)) - (\pi_X(E_2))$

- This equivalence *only* holds if $X$ includes the primary key or a set of attributes with the same properties (unique and not null)
9. Other properties

- $\sigma_{F_1 \lor F_2}(E) \equiv (\sigma_{F_1}(E)) \cup (\sigma_{F_2}(E))$
- $\sigma_{F_1 \land F_2}(E) \equiv (\sigma_{F_1}(E)) \cap (\sigma_{F_2}(E))$
10. Distribution of join with respect to union

- \( E \bowtie (E_1 \cup E_2) \equiv (E \bowtie E_1) \cup (E \bowtie E_2) \)

All binary operators are commutative and associative except for difference
Example

▷ Tables

EMP (Emp#, ......., Dept#, Salary)
DEPT (Dept#, DName, ............)

▷ SQL query

SELECT DISTINCT DName
FROM EMP, DEPT
WHERE EMP.Dept# = DEPT.Dept#
AND Salary > 1000;
Example: Algebraic transformations

\[ \pi_{\text{DName}} \left( \sigma_{\text{EMP.Dept#} = \text{DEPT.Dept#} \land \text{Salary} > 1000} \left( \text{EMP} \times \text{DEPT} \right) \right) \]
Example: Algebraic transformations

\[ \pi_{\text{DName}} \left( \sigma_{\text{EMP.Dept#} = \text{DEPT.Dept#} \land \text{Salary} > 1000} \right) (\text{EMP} \times \text{DEPT}) \]

Prop #1

\[ \pi_{\text{DName}} (\sigma_{\text{Salary} > 1000} (\sigma_{\text{EMP.Dept#} = \text{DEPT.Dept#}} (\text{EMP} \times \text{DEPT}))) \]
Example: Algebraic transformations

\[ \pi_{\text{DName}} \left( \sigma_{\text{EMP.Dept#}=\text{DEPT.Dept#} \land \text{Salary} > 1000} \ (\text{EMP} \times \text{DEPT}) \right) \]

Prop #1

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} \left( \sigma_{\text{EMP.Dept#}=\text{DEPT.Dept#}} \ (\text{EMP} \times \text{DEPT}) \right)) \]

Prop #5

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} \ (\text{EMP} \bowtie \text{DEPT})) \]
Example: Algebraic transformations

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} (\text{EMP} \bowtie \text{DEPT})) \]

Prop #3

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} (\text{EMP})) \bowtie \text{DEPT} \]
Example: Algebraic transformations

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} (\text{EMP} \bowtie\!\!\!\!\!\!\text{DEPT})) \]

Prop #3

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} (\text{EMP})) \bowtie\!\!\!\!\!\!\text{DEPT} \]

Prop #2 and #4

\[ \pi_{\text{DName}}((\pi_{\text{Dept#}}(\sigma_{\text{Salary} > 1000}(\text{EMP})) \bowtie\!\!\!\!\!\!\text{DEPT})) \]
Example: Query tree

Final query tree

\[
\begin{align*}
\text{σ}_{\text{Salary} > 1000} & \quad \text{DEPT} \\
\text{π}_{\text{Dept}#} & \quad \text{π}_{\text{Dept}#, \text{DName}} \\
\text{EMP} & \quad \text{π}_{\text{DName}} \\
\end{align*}
\]
Example: Cardinalities

- Cardinality (EMP) ≈ 10,000
- Cardinality (DEPT) ≈ 100
- Cardinality (EMP where Salary > 1000) ≈ 50
Database Management Systems

Cost based optimization
Cost based optimization

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

"CANONICAL" QUERY TREE

COST BASED OPTIMIZATION

DATA DICTIONARY

DATA PROFILES (STATISTICS ON DATA)

ACCESS PROGRAM

SET OF DEPENDENCIES
Cost based optimization

It is based on

- **Data profiles**
  - statistical information describing data distribution for tables and intermediate relational expressions

- **Approximate cost formulas for access operations**
  - Allow evaluating the cost of different alternatives for executing a relational operator
Database Management Systems

Data profiles
Table profiles

Quantitative information on the characteristics of tables and columns

- cardinality (# of tuples) in each table $T$
  - also estimated for intermediate relational expressions
- size in bytes of tuples in $T$
- size in bytes of each attribute $A_j$ in $T$
- number of distinct values of each attribute in $T$
  - cardinality of the active domain of the attribute
- min and max values of each attribute $A_j$ in $T$
Table profiles are stored in the data dictionary

Profiles should be periodically refreshed by re-analyzing data in the tables

- Update statistics command
- Executed on demand
  - Immediate execution during transaction processing would overload the system
Table profiles are exploited to estimate the size of intermediate relational expressions

- For the selection operator

\[
\text{Card } (\sigma_{A_i = v} (T)) \approx \text{Card } (T) / \text{Val } (A_i \text{ in } T)
\]

- \text{Val } (A_i \text{ in } T) = \# \text{ of distinct values of } A_i \text{ in } T (\text{active domain})

It holds only under the hypothesis of \textit{uniform distribution}
Database Management Systems

Access operators
Internal representation of the relational expression as a query tree

\[
\begin{align*}
\sigma_{\text{Salary} > 1000} & \quad \pi_{\text{Dept}#} \quad \pi_{\text{Dept},\text{DName}} \\
\pi_{\text{DName}} & \quad \pi_{\text{Dept}#} \\
\text{DEPT} & \quad \text{EMP}
\end{align*}
\]
Leaves correspond to the physical structures
- tables, indices

Intermediate nodes are operations on data supported by the given physical structure
- e.g., scan, join, group by
Sequential scan

- Executes sequential access to all tuples in a table
  - also called full table scan
- Operations performed during a sequential scan
  - Projection
    - discards unnecessary columns
  - Selection on a simple predicate \((A_i = v)\)
  - Sorting based on an attribute list
  - Insert, update, delete
Classical algorithms in computer science are exploited
  • e.g., quick sort
Size of data is relevant
  • memory sort
  • sort on disk
If available, it may exploit *index* access
- B+-tree, hash, or bitmap

Simple equality predicate $A_i = v$
- Hash, B+-tree, or bitmap are appropriate

Range predicate $v_1 \leq A_i \leq v_2$
- *only* B+-tree is appropriate

For predicates with *limited selectivity* full table scan is better
- if available, consider bitmap
B+-tree versus bitmap

- **B-tree**
  - $NR \times \text{Len(Pointer)}$

- **Bitmap**
  - $NR \times NK \times 1$ bit

Len(Pointer) = 4×8 bit

Conjunction of predicates $A_i = v_1 \land A_j = v_2$

- The *most selective* predicate is evaluated first
  - Table is read through the index
- Next the other predicates are evaluated on the intermediate result

**Optimization**

- First compute the *intersection* of bitmaps or RIDs coming from available indices
- Next table read and evaluation of remaining predicates
Example: Predicate evaluation

Which female students living in Piemonte are exempt from enrollment fee?

<table>
<thead>
<tr>
<th>RID</th>
<th>Gender</th>
<th>Exempt</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
<td>Y</td>
<td>Piemonte</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>Y</td>
<td>Liguria</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
<td>N</td>
<td>Puglia</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>N</td>
<td>Sicilia</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>Y</td>
<td>Piemonte</td>
</tr>
</tbody>
</table>

RID 5
Disjunction of predicates $A_i = v_1 \lor A_j = v_2$

- Index access can be exploited *only* if all predicates are supported by an index
- otherwise full table scan
A critical operation for a relational DBMS

- connection between tables is based on values
- instead of pointers
- size of the intermediate result is typically larger than the smaller table

Different join algorithms

- Nested loop
- Merge scan join
- Hash join
- Bitmapped join
Nested loop

<table>
<thead>
<tr>
<th>Outer table</th>
<th>Inner table</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- External scan
- Internal or direct scan
- Join attribute
A single full scan is done on the outer table

For each tuple in the outer table

- a full scan of the inner table is performed, looking for corresponding values

Also called “brute force”
Nested loop

Efficient when

- inner table is small and fits in memory
  - optimized scan
- join attribute in the inner table is indexed
  - index scan

Execution cost

- The nested loop join technique is *not symmetric*
- The execution cost depends on which table takes the role of inner table
Merge scan

Left table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>scan</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

Right table

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

join attribute
Both tables are sorted on the join attributes

The two tables are scanned in parallel
  - tuple pairs are generated on corresponding values

Execution cost
  - The merge scan technique is symmetric
  - requires sorting both tables
    - may be sorted by a previous operation
    - may be read through a clustered index on join attributes

More used in the past
  - efficient for large tables, because sorted tables may be stored on disk
Hash Join

From left table

<table>
<thead>
<tr>
<th>BUCKETS for left table</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>j</td>
</tr>
<tr>
<td>p</td>
</tr>
</tbody>
</table>

From right table

<table>
<thead>
<tr>
<th>BUCKETS for right table</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>w</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>m</td>
</tr>
<tr>
<td>j</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

HASH(a)

Join Attribute
Hash join

- Application of the same hash function to the join attributes in both tables
  - Tuples to be joined end up in the same buckets
    - Collisions are generated by tuples yielding the same hash function result with different attribute value
  - A local sort and join is performed into each bucket

- Very fast join technique
Bitmapped join index

- Bit matrix that precomputes the join between two tables A and B
  - One column for each RID in table A
  - One row for each RID in table B
- Position \((i, j)\) of the matrix is
  - 1 if tuple with RID \(j\) in table A joins with tuple with RID \(i\) in table B
  - 0 otherwise
- Updates may be slow

<table>
<thead>
<tr>
<th>RID</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
Typically used in OLAP queries
- joining several tables with a large central table

Example
- Exam table, joined to Student and Course tables

Exploits one or more bitmapped join indices
- One for each pair of joined tables

Access to the large central table is the last step
Complex queries may exploit jointly:

- bitmapped join indices
- bitmap indices for predicates on single tables
Example: Bitmapped join

Average score of male students for exams of courses in the first year of the master degree

- STUDENT (Reg#, SName, Gender)
- COURSE (Course#, CName, CourseYear)
- EXAM (Reg#, Course#, Date, Grade)

```sql
SELECT AVG (Grade)
FROM STUDENT S, EXAM E, COURSE C
WHERE E.Reg# = S.Reg#
AND E.Course# = C.Course#
AND CourseYear = '1M'
AND Gender = 'M';
```
... FROM EXAM E, COURSE C
WHERE E.Course# = C.Course#
AND CourseYear = '1M' ...

Bitmap for CourseYear attribute

<table>
<thead>
<tr>
<th>RID</th>
<th>...</th>
<th>1M</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

Bitmapped join for Course-Exams join

<table>
<thead>
<tr>
<th>RID</th>
<th>1</th>
<th>...</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Bitmapped join index

1 OR 4 = 68

RIDs 1 and 4
Bitmapped join

bitmap for Course-Exam predicates and join

bitmap for Student-Exam predicates and join

AND

RIDs of Exam table for tuples to be read
Group by

- **Sort based**
  - Sort on the group by attributes
  - Next compute aggregate functions on groups

- **Hash based**
  - Hash function on the group by attributes
  - Next sort each bucket and compute aggregate functions

- **Materialized views** may be exploited to improve the performance of aggregation operations
Database Management Systems

Execution plan selection
Cost based optimization

- **Inputs**
  - Data profiles
  - Internal representation of the query tree

- **Output**
  - “Optimal” query execution plan
  - Set of dependencies

- It evaluates the cost of different alternatives for
  - reading each table
  - executing each relational operator

- It exploits approximate cost formulas for access operations
The search for the optimal plan is based on the following dimensions:

- The way data is read from disk
  - e.g., full scan, index
- The execution order among operators
  - e.g., join order between two join operations
- The technique by means of which each operator is implemented
  - e.g., the join method
- When to perform sort (if sort is needed)
The optimizer builds a *tree of alternatives* in which

- each internal node makes a decision on a variable
- each leaf represents a complete query execution plan
Example

Given 3 tables
- R, S, T

Compute the join
\[
R \bowtie S \bowtie T
\]

Execution alternatives
- 4 join techniques to evaluate (for both joins)
- 3 join orders
- In total, at most
  - \( 4 \times 4 \times 3 = 48 \) different alternatives
Example
Best execution plan selection

▷ The optimizer selects the leaf with the lowest cost

▷ General formula

\[ C_{\text{Total}} = C_{\text{I/O}} \times n_{\text{I/O}} + C_{\text{cpu}} \times n_{\text{cpu}} \]

- \( n_{\text{I/O}} \) is the number of I/O operations
- \( n_{\text{cpu}} \) is the number of CPU operations

▷ The selection is based on operation research optimization techniques

- e.g., branch and bound
The final execution plan is an approximation of the best solution.

The optimizer looks for a solution which is of the same order of magnitude of the “best” solution:

- For compile and go:
  - it stops when the time spent in searching is comparable to the time required to execute the current best plan.