Database Systems
07 Physical Design & Tuning

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management
Announcements/Org

- **#1 Video Recording**
  - Since lecture 03, video/audio recording
  - Link in TeachCenter & TUbe (video recorder fixed?)

- **#2 Statistics Exercise 1**
  - All submissions accepted (submitted/draft)
  - In progress of grading, but understaffed

- **#3 New Exercise Rules**
  - Still mandatory exercises (final exam admission)
  - Relaxed 50% successful completion rule: 1 exercise 2%-50% allowed

- **#4 Exercise 2**
  - Various issues (data cleaning, schema, etc), latest update Apr 22
  - Modified deadline: Apr 30 → May 07 11.59pm

65.3% → 77.4%

Ex.1 Sample / Feedback
Physical Design, and why should I care?

- **Performance Tuning via Physical Design**
  - Select physical data structures for relational schema and query workload
  - #1: User-level, *manual physical design* by DBA (database administrator)
  - #2: User/system-level *automatic physical design* via advisor tools

- **Example**
  
  ```sql
  SELECT * FROM R, S, T
  WHERE R.c = S.d AND S.e = T.f
  AND R.b BETWEEN 12 AND 73
  ```

  ![Diagram showing logical and physical structures of tables, views, partitioning, and access paths with B+ Tree, Bitmap, Hash, and Compression options.](image)
Agenda

- Compression Techniques
- Index Structures
- Table Partitioning
- Materialized Views
Compression Techniques
Overview Database Compression

- **Background: Storage System**
  - Buffer and storage management (incl. I/O) at granularity of **pages**
  - PostgreSQL default: **8KB**
  - Different table/page layouts (e.g., NSM, DSM, PAX, column)

- **Compression Overview**
  - **Fit larger datasets in memory**, less I/O, better cache utilization
  - Some allow query processing directly **on the compressed data**
  - #1 Page-level compression (general-purpose GZIP, Snappy, LZ4)
  - #2 Row-level heavyweight/lightweight compression
  - #3 **Column-level lightweight compression**
  - #4 Specialized log and index compression

Lightweight Database Compression Schemes

- **Null Suppression**
  - Compress integers by omitting leading zero bytes/bits (e.g., NS, gamma)

- **Run-Length Encoding**
  - Compress sequences of equal values by runs of (value, start, run length)

- **Dictionary Encoding**
  - Compress column w/ few distinct values as pos in dictionary (→ code size)

- **Delta Encoding**
  - Compress sequence w/ small changes by storing deltas to previous value

- **Frame-of-Reference Encoding**
  - Compress values by storing delta to reference value (outlier handling)
Index Structures
Overview Index Structures

- **Table Scan vs Index Scan**
  - For highly selective predicates, index scan **asymptotically much better** than table scan
  - Index scan **higher per tuple overhead** (break even ~5% output ratio)
  - Multi-column predicates: fetch/RID-list intersection

- **Uses of Indexes**
  - **Lookups / Range Scans**
    - Contains key 107?
  - **Unique Constraints**
    - ix
  - **Index Nested Loop Joins**
    - size = 7100
  - **Aggregates**
    - (count, min/max)
Classification of Index Structures

- Traditional Classification
  - 1D Access Methods
    - Key Comparison
      - Sequential Lists
        - Linked Lists
      - Tree-Based
        - Binary Search Trees
          - Multiway Trees (B-Tree)
          - Prefix Trees (Tries)
      - Sort-Based
      - Hash-Based
        - Static
        - Dynamic

- Prefix Trees for in-memory DBs
  - [Matthias Boehm et al: Efficient In-Memory Indexing with Generalized Prefix Trees. BTW 2011]
  - [Viktor Leis, Alfons Kemper, Thomas Neumann: The adaptive radix tree: ARTful Indexing for Main-Memory Databases. ICDE 2013]
Recap: Index Creation/Deletion via SQL

**Create Index**
- Create a secondary (nonclustered) index on a set of attributes
  - **Clustered** (primary): tuples sorted by index
  - **Non-clustered** (secondary): sorted attribute with RIDs
  - Can specify uniqueness, order, and indexing method
  - **PostgreSQL:** [btree], hash, gist, and gin

```
CREATE INDEX ixStudLname ON Students USING btree
(Lname ASC NULLS FIRST);
```

**Delete Index**
- Drop indexes by name

```
DROP INDEX ixStudLname;
```

**Tradeoffs**
- Indexes often automatically created for primary keys / unique attributes
- **Lookup.scan.join performance** vs **insert performance**
- Analyze usage statistics: pg_stat_user_indexes, pg_stat_all_indexes
B-Tree Overview

- **History B-Tree**
  - Bayer and McCreight 1972 (multiple papers), Block-based, Balanced, Boeing
  - Multiway tree (node size = page size); designed for DBMS
  - Extensions: B+-Tree/B*-Tree (data only in leafs, double-linked leaf nodes)

- **Definition B-Tree (k, h)**
  - All paths from root to leafs have equal length $h$
  - All nodes (except root=leaf) have $[k, 2k]$ key entries
  - All nodes (except root, leafs) have $[k+1, 2k+1]$ successors
  - Data is a record or a reference to the record (RID)

$\log_{2k+1}(n+1) \leq h \leq \left\lfloor \log_{k+1}\left(\frac{n+1}{2}\right) \right\rfloor + 1$

![Diagram of B-Tree](image)
B-Tree Search

- **Example B-Tree k=2**
  - Get 38 → D38
  - Get 20 → D20
  - Get 6 → NULL

- **Lookup $Q_K$ within a node**
  - Scan / binary search keys for $Q_K$, if $K_i = Q_K$, return $D_i$
  - If node does not contain key
    - If leaf node, abort search w/ NULL (not found), otherwise
    - Decent into subtree $P_i$ with $K_i < Q_K ≤ K_{i+1}$

- **Range Scan $Q_L < K < U$**
  - Lookup $Q_L$ and call next $K$ while $K < Q_U$ (keep current position and node stack)
B-Tree Insert

- **Basic Insertion Approach**
  - Always insert into leaf nodes!
  - Find position similar to lookup, insert and maintain sorted order
  - If node overflows (exceeds 2k entries) ➔ node splitting

- **Node Splitting Approach**
  - Split the 2k+1 entries into two leaf nodes
  - **Left node**: first k entries
  - **Right node**: last k entries
  - (k+1)th entry inserted into parent node ➔ can cause **recursive splitting**
  - Special case: root split (h++)

- **B-Tree is self-balancing**
B-Tree Insert, cont. (Example w/ k=1)

- Insert 1
- Insert 5
- Insert 2 (split)
- Insert 6
- Insert 7 (split)
- Insert 4
- Insert 8
- Insert 3 (2x split)

Note: Exercise 03, Task 3.2 (B-tree insertion and deletion)
B-Tree Delete

- **Basic Deletion Approach**
  - Lookup deletion key, abort if non-existing
  - Case inner node: move entry from fullest successor node into position
  - Case leaf node: if underflows (<k entries) ➔ merge w/ sibling

- **Example**
  - Case inner
    - [Diagram]
  - Case leaf
    - [Diagram]
Prefix Trees (Radix Trees, Tries)

- **Generalized Prefix Tree**
  - Arbitrary data types (byte sequences)
  - Configurable prefix length $k'$
  - Node size: $s = 2^{k'}$ references
  - Fixed maximum height $h = k/k'$
  - Secondary index structure

- **Characteristics**
  - Partitioned data structure
  - Order-preserving (for range scans)
  - Update-friendly

- **Properties**
  - Deterministic paths
  - Worst-case complexity $O(h)$

Prefix Trees (Radix Trees, Tries)
Excursus: Learned Index Structures

- **A Case For Learned Index Structures**
  - Sorted data array, predict position of key
  - **Hierarchy of simple models** (stages models)
  - Tries to approximate the CDF similar to interpolation search (uniform data)

- **Follow-up Work on SageDBMS**

[Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis: The Case for Learned Index Structures. SIGMOD 2018]

Table Partitioning
Overview Partitioning Strategies

- **Horizontal Partitioning**
  - Relation partitioning into disjoint subsets

- **Vertical Partitioning**
  - Partitioning of attributes with similar access pattern

- **Hybrid Partitioning**
  - Combination of horizontal and vertical fragmentation (hierarchical partitioning)

- **Derived Horizontal Partitioning**
Correctness Properties

- **#1 Completeness**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation $R$ is partitioned into $n$ fragments)
  - Each item from $R$ must be included in at least one fragment

- **#2 Reconstruction**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation $R$ is partitioned into $n$ fragments)
  - Exact reconstruction of fragments must be possible

- **#3 Disjointness**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation $R$ is partitioned into $n$ fragments)
  - $R_i \cap R_j = \emptyset$ ($1 \leq i, j \leq n; i \neq j$)
Horizontal Partitioning

- **Row Partitioning into n Fragments** $R_i$
  - Complete, disjoint, reconstructable
  - Schema of fragments is equivalent to schema of base relation

- **Partitioning**
  - Split table by n selection predicates $P_i$ (partitioning predicate) on attributes of $R$
  - Beware of attribute domain and skew

- **Reconstruction**
  - Union of all fragments
  - Bag semantics, but no duplicates across partitions

$$R_i = \sigma_{P_i}(R) \quad (1 \leq i \leq n)$$

$$R = \bigcup_{1 \leq i \leq n} R_i$$
Vertical Fragmentation

- **Column Partitioning into n Fragments Ri**
  - Complete, reconstructable, but not disjoint
    (primary key for reconstruction via join)
  - Completeness: each attribute must be included in at least one fragment

- **Partitioning**
  - Partitioning via projection
  - Redundancy of primary key

- **Reconstruction**
  - Natural join over primary key

- **Hybrid horizontal/vertical partitioning**

\[
R_i = \pi_{PK,A_i}(R) \\
(1 \leq i \leq n)
\]

\[
R = R_1 \bowtie R_i \bowtie R_n \\
(1 \leq i \leq n)
\]

\[
R = R_1 \bowtie R_i \bowtie R_n \text{ w/ } R_i = \bigcup R_{ij} \\
\Rightarrow R = \bigcup R_j \text{ w/ } R_j = R_{1j} \bowtie R_{ij} \bowtie R_{nj}
\]
Derived Horizontal Fragmentation

- **Row Partitioning** $R$ into $n$ fragments $R_i$, with partitioning predicate on $S$
  - Potentially complete (not guaranteed), **restructable, disjoint**
  - Foreign key / primary key relationship determines correctness

- **Partitioning**
  - **Selection** on independent relation $S$
  - **Semi-join** with dependent relation $R$
    to select partition $R_i$

- **Reconstruction**
  - Equivalent to horizontal partitioning
  - **Union** of all fragments

\[
R_i = R \bowtie S_i = R \bowtie \sigma_{P_i}(S) = \pi_{R,*} \left( R \bowtie \sigma_{P_i}(S) \right)
\]

\[
R = \bigcup_{1 \leq i \leq n} R_i
\]
Exploiting Table Partitioning

- Partitioning and query rewriting
  - #1 Manual partitioning and rewriting
  - #2 Automatic rewriting (spec. partitioning)
  - #3 Automatic partitioning and rewriting

Example PostgreSQL (#2)

```sql
CREATE TABLE Squad(
    JNum INT PRIMARY KEY,
    Pos CHAR(2) NOT NULL,
    Name VARCHAR(256)
) PARTITION BY RANGE(JNum);

CREATE TABLE Squad10 PARTITION OF Squad
    FOR VALUES FROM (1) TO (10);

CREATE TABLE Squad20 PARTITION OF Squad
    FOR VALUES FROM (10) TO (20);

CREATE TABLE Squad24 PARTITION OF Squad
    FOR VALUES FROM (20) TO (24);
```
Example, cont.

SELECT *  FROM  Squad
WHERE  JNum > 11 AND JNum < 20

\[ \sigma_{\text{JNum}>11 \land \text{JNum}<20} \]

\[ \cup \]

\[ \sigma_{\text{JNum}>11 \land \text{JNum}<20} \]

S10  S20  S24

\[ \sigma_{\text{JNum}>11 \land \text{JNum}<20} \]

\[ \cup \]

\[ \sigma_{\text{JNum}>11 \land \text{JNum}<20} \]

S24

\[ \cup \]

\[ \sigma_{\text{JNum}>11 \land \text{JNum}<20} \]

S10  S20

JNum in [1,10)  JNum in [10,20)

JNum in [20,24)
Excursus: Database Cracking

- **Core Idea:** Queries trigger physical reorganization (partitioning and indexing)

[Stratos Idreos, Martin L. Kersten, Stefan Manegold: Database Cracking. **CIDR 2007**]

<table>
<thead>
<tr>
<th>A</th>
<th>A_{CRK}</th>
<th>Q_1: \sigma_{A&gt;5 \land A&lt;10}</th>
<th>Q_2: \sigma_{A&gt;2 \land A&lt;15}</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>3</td>
<td>\leq 5</td>
<td>{\leq 2}</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>&gt; 5</td>
<td>&gt; 2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>\geq 10</td>
<td>\geq 5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**#1 Automatic Partitioning**

**#2 AVL/B-tree over Partitions**
Materialized Views
Overview Materialized Views

- Core Idea of Materialized Views
  - Identification of frequently re-occurring queries (views)
  - Precompute subquery results once, store and reuse many times

- The MatView Lifecycle
  - #1 View Selection
    (automatic selection via advisor tools, approximate algorithms)
  - #2 View Usage
    (transparent query rewrite for full/partial matches)
  - #3 View Maintenance
    (maintenance time and strategy, when and how)
Materialized Views

View Selection and Usage

- **Motivation**
  - Shared subexpressions very common in analytical workloads
  - Ex. Microsoft’s Analytics Clusters

- **#1 View Selection**
  - Exact view selection (query containment) is **NP-hard**
  - Heuristics, greedy and approximate algorithms

- **#2 View Usage**
  - Given query and set of materialized view, decide which views to use and rewrite the query for produce correct results
  - Generation of compensation plans


View Maintenance – When?

- **Materialized view creates redundancy → Need for #3 View Maintenance**

- **Eager Maintenance (writer pays)**
  - Immediate refresh: updates are directly handled (consistent view)
  - On Commit refresh: updates are forwarded at end of successful TXs

- **Deferred Maintenance (reader pays)**
  - Maintenance on explicit user request
  - Potentially *inconsistent base tables and views*

- **Lazy Maintenance (async/reader pays)**
  - Same guarantees as eager maintenance
  - Defer maintenance until free cycles or view required (invisible for updates and queries)

Materialized Views

View Maintenance – How?

- Incremental Maintenance
  - Propagate: Compute required updates
  - Apply: apply collected updates to the view

Example View:
```
SELECT A, SUM(B) FROM Sales GROUP BY CUBE(A)
```

<table>
<thead>
<tr>
<th>A</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>107</td>
</tr>
<tr>
<td>X</td>
<td>30</td>
</tr>
<tr>
<td>Y</td>
<td>77</td>
</tr>
</tbody>
</table>

Global Net Delta \( \Delta R \)

Local View Delta \( \Delta V_L \)

[Global View Delta] \( \Delta V_G \)

Super Delta \( \Delta V_S \)

Apply Delta \( \Delta V_A \)

A | B | A | SUM |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+ NULL</td>
<td>3</td>
</tr>
<tr>
<td>+ X</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ NULL</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ Z</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Incremental Propagate

Incremental Apply
Materialized Views in PostgreSQL

- **View Selection**
  - Manual definition of materialized view only
  - With or without data

- **View Usage**
  - Manual use of view
  - No automatic query rewriting

- **View Maintenance**
  - Manual (deferred) refresh
  - Complete, no incremental maintenance
  - Note: Community work on IVM

```
CREATE MATERIALIZED VIEW TopScorer
AS
SELECT P.Name, Count(*)
FROM Players P, Goals G
WHERE P.Pid=G.Pid AND G.GOwn=FALSE
GROUP BY P.Name
ORDER BY Count(*) DESC
WITH DATA;

REFRESH MATERIALIZED VIEW TopScorer;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>James Rodríguez</td>
<td>6</td>
</tr>
<tr>
<td>Thomas Müller</td>
<td>5</td>
</tr>
<tr>
<td>Robin van Persie</td>
<td>4</td>
</tr>
<tr>
<td>Neymar</td>
<td>4</td>
</tr>
<tr>
<td>Lionel Messi</td>
<td>4</td>
</tr>
<tr>
<td>Arjen Robben</td>
<td>3</td>
</tr>
</tbody>
</table>

[Yugo Nagata: Implementing Incremental View Maintenance on PostgreSQL, PGConf 2018]
Conclusions and Q&A

- **Summary**
  - **Physical Access Paths**: Compression and Index Structures
  - **Logical Access Paths**: Table Partitioning and Materialized Views

- **Exercise 2 Reminder**
  - Submission deadline: Apr 30 → **May 07 11.59pm**
  - **Start early** (most time consuming of all four exercises)

- **Next Lectures**
  - May 6: **08 Query Processing**
  - May 13: **09 Transaction Processing and Concurrency**