Automatic Physical DB Design

Outline

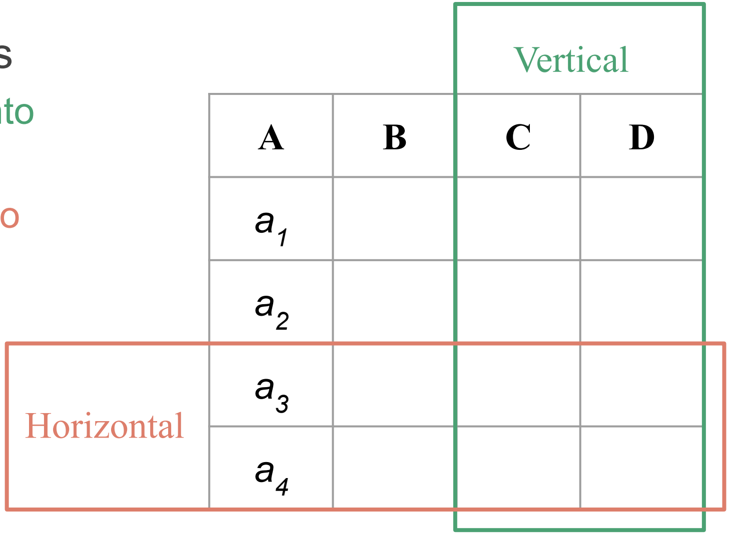
- Introduction
- Vertical Partitioning
- Horizontal Partitioning
- Conclusions

Introduction

Physical Design Features

- Partitioning: logic relation → physical tables
  - Vertical Partitioning: subdividing ***attributes*** into physical groups
  - Horizontal Partitioning: subdividing ***tuples*** into physical groups
- Replication
- Sort Order
- Encoding/Compression

- Index Selection
  - Vertical + Replication + Sort [+ Horizontal]
- Materialized View



Vertical Partitioning: An Example

- Given a logical relation **R(A, B, C, D)** and a set of queries:
  - Q_1: `SELECT A, B FROM R`
  - Q_2: `SELECT C, D FROM R`
- How to assign **R** to physical tables/files?
  - Option **#1**: store all attributes together, i.e., **P(A, B, C, D)**
    - Q_1 will scan **redundant** attributes **C** and **D**
    - Q_2 will scan **redundant** attributes **A** and **B**
  - Option **#2**: store each attribute separately, i.e., **P_1(A), P_2(B), P_3(C), P_4(D)**
    - Q_1: `SELECT A, B FROM P1 JOIN P2 USING rowid`
    - Q_2: `SELECT C, D FROM P3 JOIN P4 USING rowid`
  - Option **#3**: **P_1(A, B), P_2(C, D)**
    - Q_1: `SELECT * FROM P1` Q_2: `SELECT * FROM P2`
    - *What if Q_3: `SELECT B, C FROM R` ?*

Horizontal Partitioning: Background

- Environment: **share-nothing parallel databases**
  - tables are partitioned across nodes to enable parallel processing
    - choose a ***partitioning key*** for each table
    - perform ***hash*** *partitioning* or ***range*** *partitioning* on these keys
- Query processing: avoid **communication overhead**
  - assume two tables, **R** and **S**, are joined
    - ***local join***: **R** and **S** are **both partitioned** on the join key
    - ***broadcast join***: **R** is partitioned on the join key, **S** is **replicated** to all nodes
    - ***directed join***: **R** is partitioned on the join key, **S** is **repartitioned** on the join key
    - ***repartitioned join***: **R** and **S** are **both repartitioned** on the join keys
  - preference: *local join* > *broadcast join* or *directed join* > *repartitioned join*
  - also, local group-by, local window functions

Horizontal Partitioning: An Example

- Given a TPC-H database and a set of queries:
  - Q_1: … `lineitem JOIN orders    ON l_orderkey = o_orderkey`
  - Q_2: … `lineitem JOIN supplier ON l_suppkey = s_suppkey`
- How to choose partitioning keys for **lineitem**, **orders** and **supplier**?
  - Option #1: **lineitem(l_orderkey), orders(o_orderkey), supplier(s_suppkey)**
    - Tuples with the same *orderkey* are assigned to the same node
    - Q_1: local join Q_2: redirected join
  - Option #2: **lineitem(l_suppkey), orders(o_orderkey), supplier(s_suppkey)**
    - Tuples with the same *suppkey* are assigned to the same node
    - Q_1: redirected join Q_2: local join

Other Physical Design Features

- Replication
  - Trade-off: ***storage cost <-> {performance, safety}***
  - e.g., *broadcast join* vs *local join*
- Sort Order
  - ***ORDER BY, sort merge join, pipelined group-by***
- Encoding/Compression
  - Trade-off: ***{storage, I/O} cost <-> CPU cost***
  - *query evaluation on encoded/compressed data*

Other Physical Design Features

- Replication
  - Trade-off: ***storage cost <-> {performance, safety}***
  - e.g., *broadcast join* vs *local join*
- Sort Order
  - ***ORDER BY, sort merge join, pipelined group-by***
- Encoding/Compression
  - Trade-off: ***{storage, I/O} cost <-> CPU cost***
  - *query evaluation on encoded/compressed data*

- This slide will focus on *vertical/horizontal partitioning*

Vertical Partitioning

- Problem Formulation
- Algorithms
- Cost Model
- Comparison with Column Stores

Problem Formulation

- Given:
  - A set of relations $\mathbf{R} = \{\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R_n}\}$
  - A query workload $\mathbf{W}$
  - A storage upper bound $\mathbf{B}$
- Generate a set of partitions $\mathbf{P} = \{\mathbf{P_1}, \mathbf{P_2}, \ldots, \mathbf{P_N}\}$ such that:
  - Each $\mathbf{P_k} \in \mathbf{P}$ stores a subset of the attributes of $\mathbf{R}$ plus the identifier attributes
  - All data in $\mathbf{R}$ fit into $\mathbf{P}$ (except for the identifier values)
  - Total storage cost of $\mathbf{P}$ does not exceed $\mathbf{B}$
  - The overall cost of $\mathbf{W}$ is minimized

- Other constraints
  - e.g., each attribute of $\mathbf{R_i}$ is contained in exactly one partition $\mathbf{P_k}$

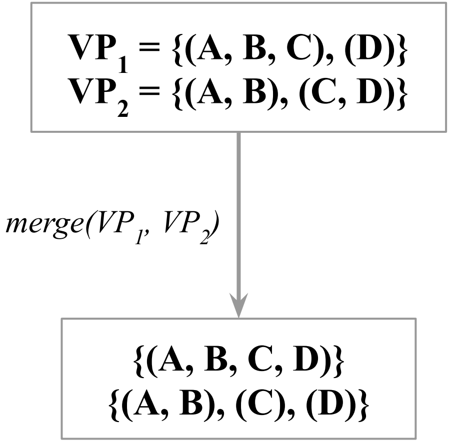
Simplified Problem

- $\mathbf{R} = \{\mathbf{R_1}\}$, where $\mathbf{R_1(A_1, A_2, \ldots, A_m)}$
- Each attribute $\mathbf{A_j}$ of $\mathbf{R_1}$ is contained in exactly one partition $\mathbf{P_k}$
- The size of solution space: *Bell number* $\mathbf{B_m}$
  - the number of different ways to partition a set, $\mathbf{O}(e^{n\ln(n)})$

- Various algorithms have been proposed [Jindal'13]
  - Basic idea #1: exploiting ***co-occurence*** of columns
  - Basic idea #2: using a ***cost model*** to evaluate the ***benefit*** of different choices
  - Top-down or bottom-up
    - Top-down: [(A, B, C, D)] → [(A, B), (C, D)] → [(A), (B), (C, D)]
    - Bottom-up: [(A), (B), (C), (D)] → [(A), (B), (C, D)] → [(A, B), (C, D)]

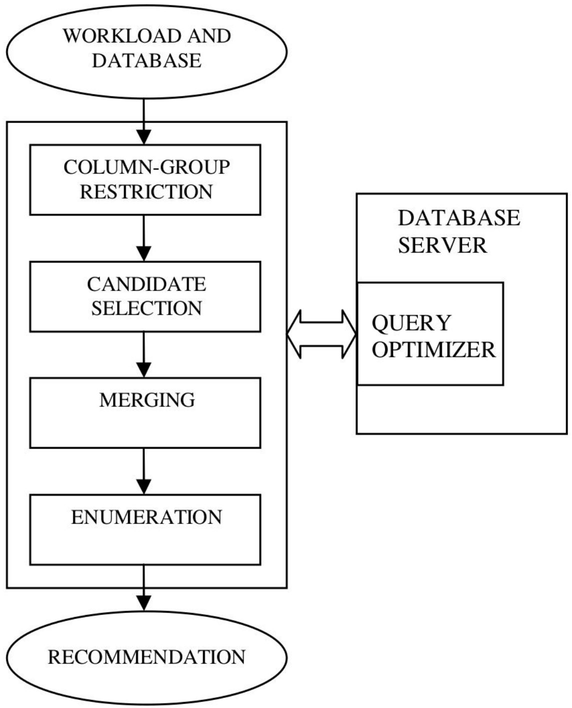
Bottom-Up Approaches

- The **Hill-Climb** algorithm [Hankins'03] (***considered the best*** [Jindal'13])
  - Starting point: *each column as a partition*
  - In each iteration, merge two partitions that the **merging** of them is most beneficial
  - Stop when there is no improvement
- Algorithm used by the **AutoAdmin** project [Agrawal'04]
  - Starting point: interesting *partition schemes* per table
    - on a per-query basis: {***frequent itemset***, remain columns}
    - support pruning + interestness pruning
  - Generate merged partition schemes
    - merge two partition schemes
    - via ***union*** or ***intersection*** of column-groups



The Cost Model

- Reinvent a new cost model?
  - 😓 Don’t *reverse-engineer* the query optimizer!
- Instead, optimizer “in the loop” [Agrawal’06]
  - ensure automated physical design is ***in-sync*** with decisions made by the optimizer
- Creating “what-if”/**FAKE** physical structures
  - query optimizer does not require physical design to be materialized
  - instead, it relies on statistics to choose right plan
  - so, build approximate statistics
  - and change the “meta-data” entry
- Workload compression: more robust cost model



Are So Many Column Stores Doing It ***Wrong?***

- Interesting **lessons learned** in [Jindal'13]
  - Vertical partitioning improves over *column layout* only for **buffer sizes < 100 MB**
  - Improvements over column layout
    - TPC-H: 3.7%, Star Schema Benchmark: 5.3%
    - Main memory: 0%
    - Commercial column-oriented DBMS: -xx%
      - Heavy compression is used for column-groups
  - ***Column layouts are often GOOD enough!***
    - Or even better

Horizontal Partitioning

- Problem Formulation
- Algorithms

Problem Formulation

- Given:
  - A database $\mathbf{D} = \{\mathbf{R1, R2, \ldots, Rn}\}$, where $\mathbf{Ri(Ai_1, Ai_2, \ldots, Ai_{Ni})}$
  - A workload $\mathbf{W}$
  - A storage bound $\mathbf{B}$
- Find a partitioning strategy $\mathbf{P}$ for $\mathbf{D}$ such that:
  - The size of replicated tables fits in $\mathbf{B}$
  - The overall cost of $\mathbf{W}$ is minimized

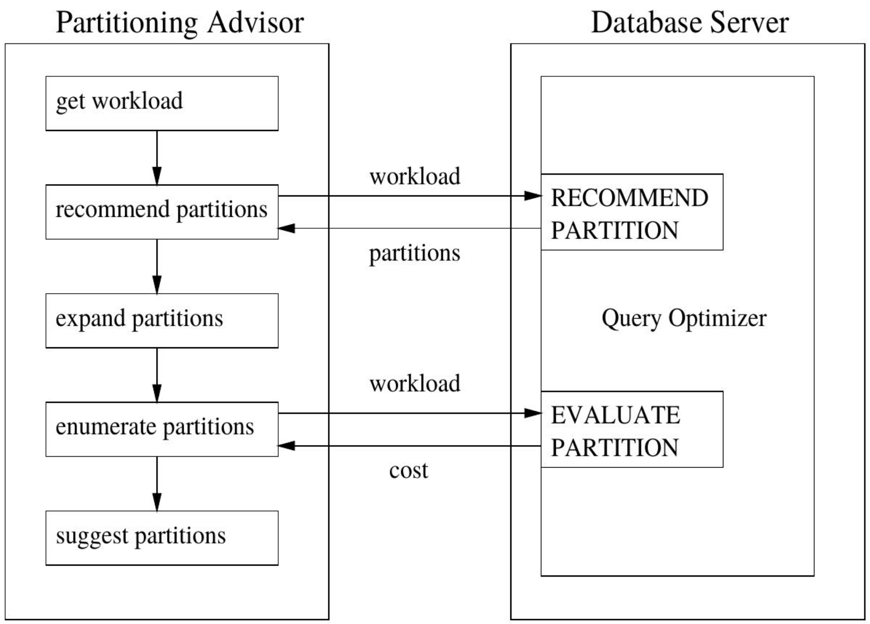
- Solution space of $\mathbf{P}$: $\mathbf{(p_1, p_2, \ldots, p_n), p_i \in \{Ai_1, Ai_2, \ldots, Ai_{Ni}, R\}}$ ($\mathbf{R}$: replication)
  - Assume only **one** column of each table is used for partitioning
  - Size: $\mathbf{(N1 + 1) * (N2 + 1) * \ldots * (Nn + 1)}$

Basic/Naive Ideas [Zilio'96]

- Elimination of "bad" attributes
  - Attributes that have **high skew** or **low column cardinality**
- Elimination of small tables
  - Small tables can be replicated to each node
- Suboptimal solution - choose the most **beneficial** partition key for each table
  - For each query, add a **weight** to underly columns according to operations
    - Join: +1.0, group-by: +0.1
    - Duplicate removal: +0.08, constant selection: -0.05, parametric selection: +0.05
  - Choose the highest weighted column
- Improved solution - consider ***α%-highest*** weighted columns
  - Exhaustively search all combinations
  - Use query optimizer to estimate the cost

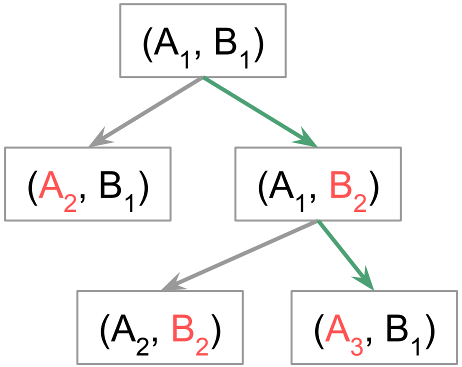
Optimizer In The Loop [Rao'02]

- RECOMMEND mode
  - For a given query, recommend **one** partition key for each base table
  - Via generating query plans for each **interesting** partitioning and estimating their cost
  - Equi-join, group-by, selection
- Expansion
  - **<T.a, T.b> $\wedge$ <T.a, T.c> = <T.a>**
- EVALUATE mode
  - Estimate the cost of queries for a given partitioning scheme



Exploring The Search Space

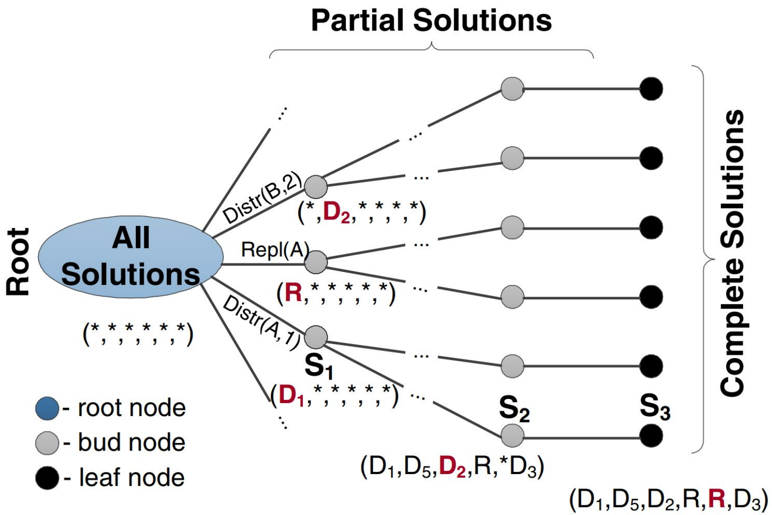
- Need to choose ***exactly one*** candidate for each table
- Exhaustive search may still be too expensive



- ***Rank-based*** search [Rao'02]
    - Root node $\mathbf{P_0}$: the most beneficial candidate for each table
    - Expand a node $\mathbf{P}$
        - Consider all child configurations that differ from $\mathbf{P}$ in exactly one $\mathbf{p_i}$
        - The different partition has the next highest ***benefit value***
    - All the expanded nodes are ranked and kept in an ordered queue
    - Choose the node with highest rank as the next search point
- Rank function: $rank(P) = -(cost(P.parent) - p_i.benefit * sqrt(Ri.size / max\_size))$

Branch and Bound Search [Nehme'11]

- ***-partitioned table***
  - The optimizer can pick any **concrete** partition scheme for query plans referencing this table
  - Used by the **bounding** function
- Branching policy
  - Node selection: depth-first
    - The first **incumbent** is reached quickly
  - Table/column selection: rank-based
    - Try replication before any partitioning
  - Pruning
    - Storage bound
    - No descendant will be optimal



The FINDER Algorithm [Garcia-Alvarado'12]

- Used by ***Greenplum***
- Begin with a randomly chosen partitioning scheme $\mathbf{P} = \mathbf{P_{best}} = \mathbf{P_0}$
- Evaluation
  - Estimate $\mathbf{cost(W, P)}$ using the optimizer
  - Record estimated ***data movement*** associated with specific column sets
  - If $\mathbf{cost(W, P) < cost(W, P_{best})}$: $\mathbf{P_{best} = P}$
- Generation
  - Find the top-k column sets that caused the most data movement
  - Create additional partitioning schemes for the next iteration

Conclusions

Conclusions

- Vertical partitioning is NOT necessary
  - Instead, **normalized schema + column store + materialized view**?

- Workload-based partitioning key selection is still meaningful

- Another aspect of horizontal partitioning: **Data Skipping**
  - Categorical partitioning: divide a table horizontally into some sub-tables according to its categorical attributes
    - **Sales → Sales-2016-01, Sales-2016-02, ...**

References

- Varadarajan R, Bharathan V, Cary A, Dave J, Bodagala S. *DBDesigner: A customizable physical design tool for Vertica Analytic Database.* ICDE 2014
- Jindal A, Palatinus E, Pavlov V, Dittrich J. *A Comparison of Knives for Bread Slicing.* PVLDB 2013
- Hankins RA, Patel JM. *Data Morphing: An Adaptive, Cache-Conscious Storage Technique.* PVLDB 2003
- Papadomanolakis S, Ailamaki A. *AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning.* SSDBM 2004
- Agrawal S, Bruno N, Chaudhuri S, Narasayya VR. *AutoAdmin: Self-Tuning Database Systems Technology.* IEEE Data Eng. Bull. 2006

References

- Zilio DC, Jhingran A, Padmanabhan S. *Partitioning Key Selection for a Shared-Nothing Parallel Database System.* IBM Research Report 1996
- Rao J, Zhang C, Lohman G, Megiddo N. *Automating Physical Database Design in a Parallel Database.* SIGMOD 2002
- Nehme R, Bruno N. *Automated Partitioning Design in Parallel Database Systems.* SIGMOD 2011
- Garcia-Alvarado C, Raghavan V, Narayanan S, Waas FM. *Automatic Data Placement in MPP Databases.* ICDEW 2012