Machine Learning for Data Management Systems

AutoAdmin

Amol Deshpande
February 2, 2023

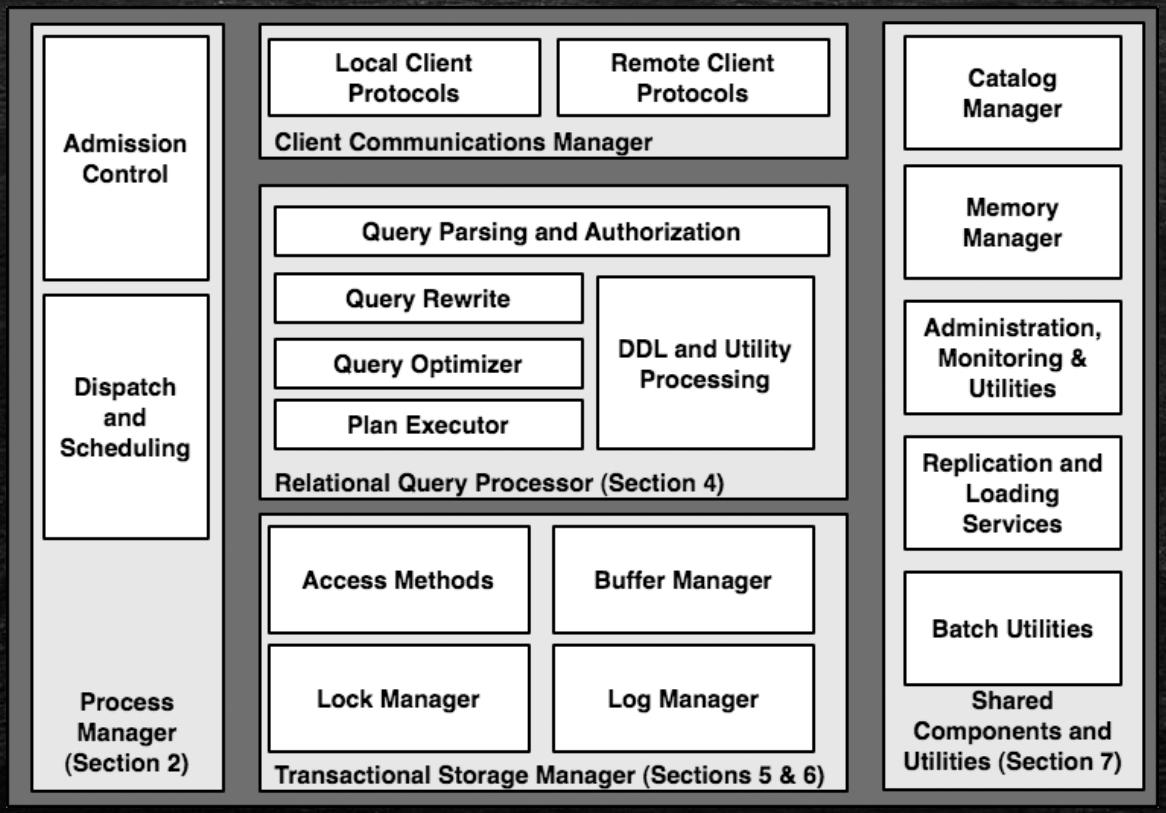
Outline

- **Architecture of Database System**
- AutoAdmin
- Towards self-driving databases

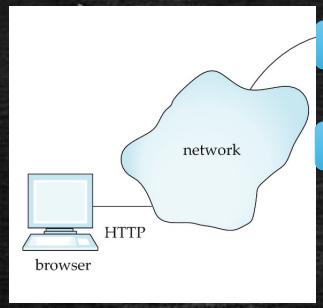
Architecture of a Traditional DBMS

- Paper by: Hellerstein, Stonebraker, Hamilton
- Covers the main components of a typical relational DBMS
- Will cover briefly for now, and revisit as appropriate later

Main Components



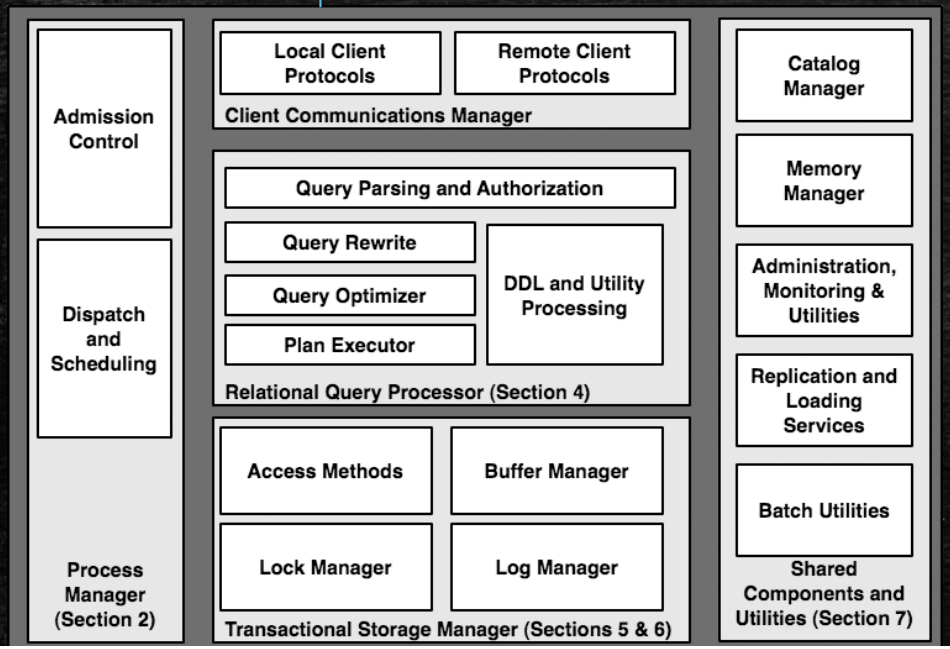
Life of a Query



Web Server

App Server

ODBC/JDBC



Clients connect using standard or proprietary protocols to submit "queries"/"transactions"

Admission Control

Assign a "thread of computation"

Parse, compile, optimize the query

Start fetching or updating the data
- get locks
- create log records if needed
- etc…

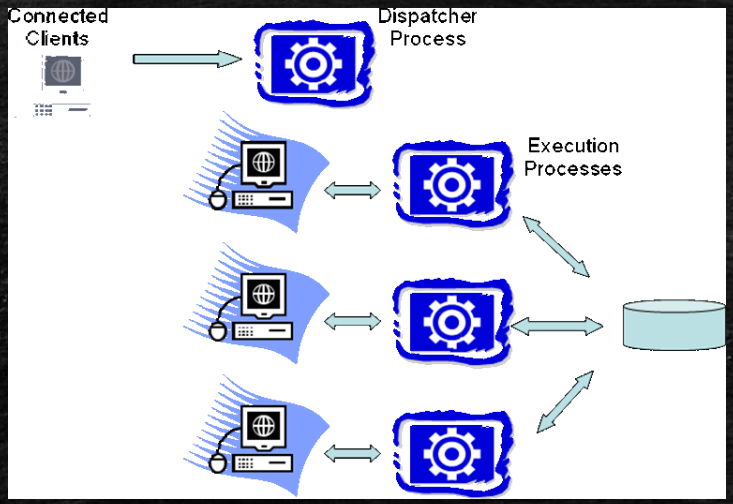
Return data batch-at-a-time

Process Models

- Question: How do we handle multiple user requests/queries “concurrently”?
- Lot of variations across Operating Systems
  - OS Process: Private address space – scheduled by kernel
  - OS (Kernel) Thread: Multiple threads per process – shared memory
    - Support for this relatively recent (late 90’s, early 00’s)
    - OS can “see” these threads and does the scheduling
  - Lightweight threads in user space
    - Scheduled by the application
    - Need to be very very careful, because OS can’t pre-empt
    - e.g., can’t do Synchronous I/O
  - DBMS Threads
    - Similar to general lightweight threads, but special-purpose

Process per DBMS Worker

- Each query gets its own "process" (e.g., PostgreSQL, IBM D2, Oracle)*
  - Heavy-weight, but easy to port to other systems
  - Need support for "shared memory" (for lock tables, etc)



How the DBMS supports concurrent queries is independent of the OS.

* All circa 2007 – may have changed since then.

Thread per DBMS Worker

- A single-multithreaded server
  - Need support for “asynchronous” I/O (so threads don’t block)
  - Easy to share state, but also makes it easy for queries to interfere

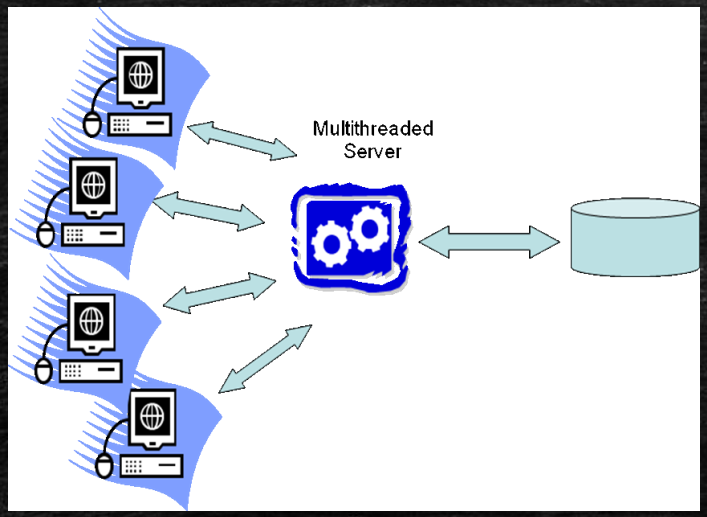


Fig. 2.2: Thread per DBMS worker model: each DBMS worker is implemented as an OS thread

Process (or Thread) Pools

- Typically DBMS allots a pool of processes or threads, and multiplexes clients/requests across those

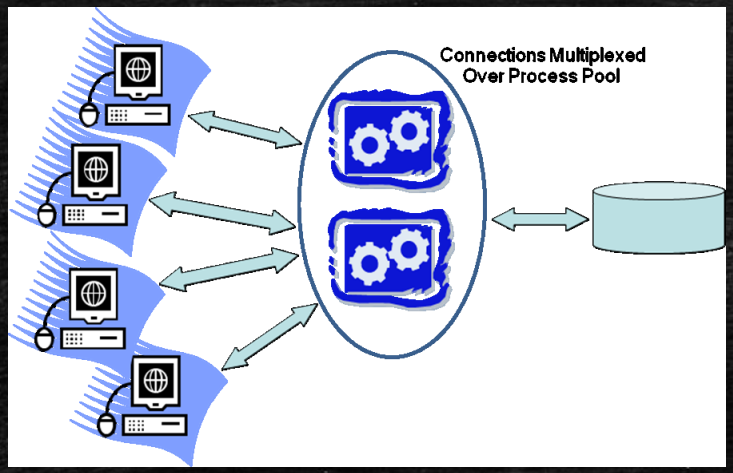
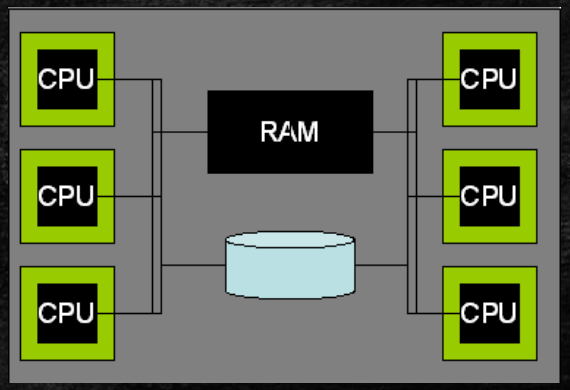
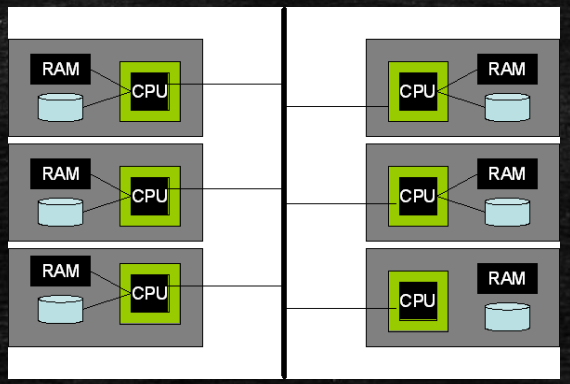


Fig. 4.2: Process Pool: each DBMS Worker is allocated to one of a pool of OS processes as work requests arrive from the Client and the process is returned to the pool once the request is satisfied.

Parallel Architectures

- Shared-memory and shared-nothing architectures prevalent today
- Shared-memory: easy to evolve to because of shared data structures
- Shared-nothing: require more coordination
  - Data must be partitioned across disks, and query processing needs to be aware of that
  - Single-machine failures need to be handled gracefully





Parallel Architectures

- Shared-disk (e.g., through use of Storage Area Networks)
  - Somewhat easier to administer, but requires specialized hardware
  - Main difference between this and shared-nothing is primarily the retrieval costs
- Non-uniform Memory Access (NUMA)
  - Seen increasingly today with many-core systems
  - Any processor can access any other processor's memory, but the costs vary

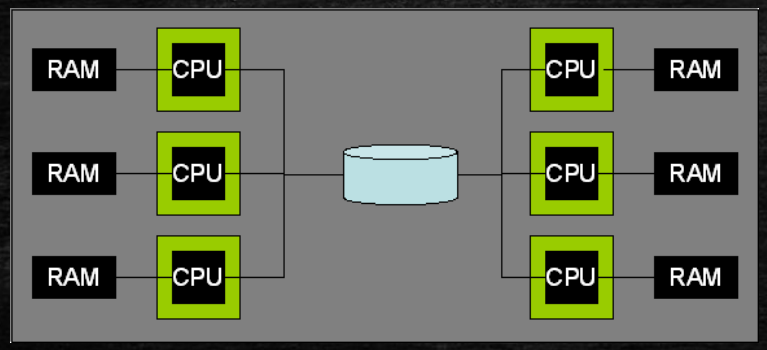
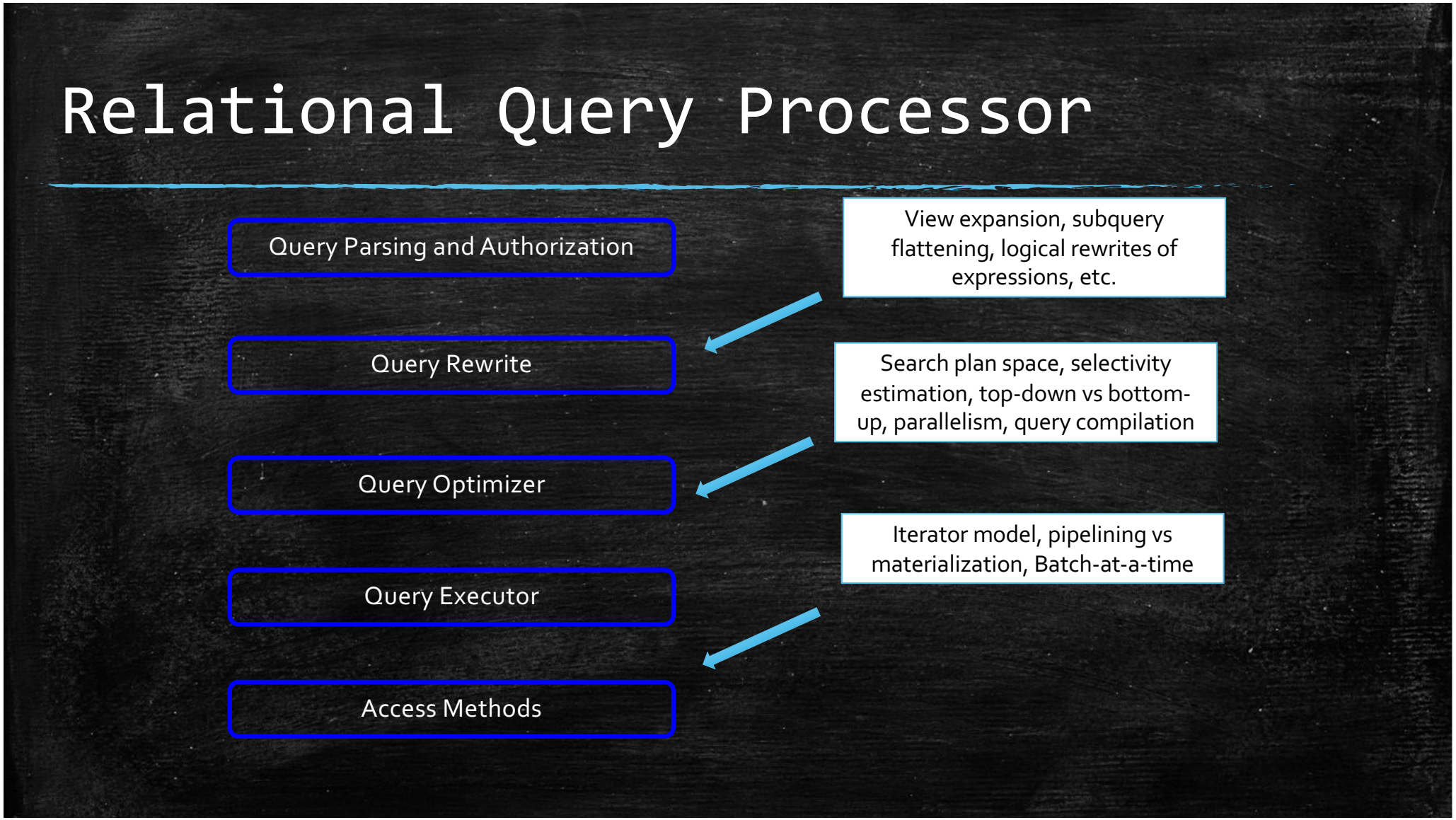
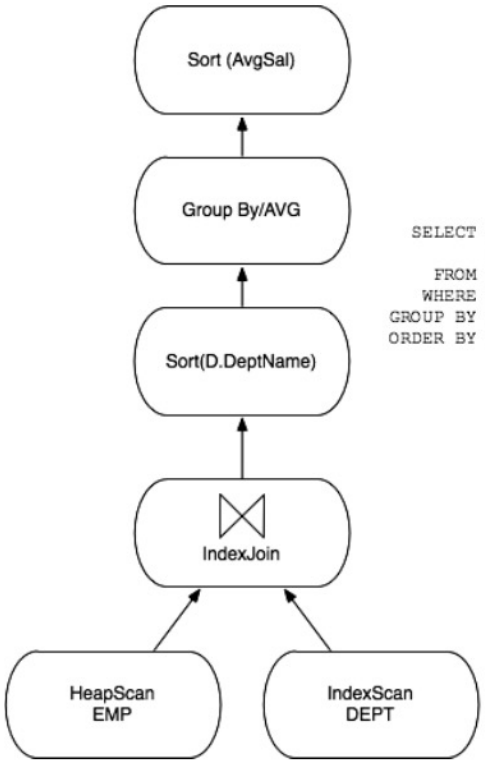


Figure 3.3 Shared-disk architecture

Relational Query Processor



A Query Plan




```
SELECT D.DeptName,
       AVG(E.Salary) AS AvgSal
  FROM EMP E, DEPT D
 WHERE E.Dno = D.DeptID
GROUP BY DeptName
ORDER BY AvgSal DESC
```

Fig. 4.1 A Query plan. Only the main physical operators are shown.

Transactions

- Atomicity
  - Need to ensure all actions of a transaction complete or not
  - Guaranteed through "logging" (i.e., maintaining a log of all operations), and using the log to "undo" or "redo" actions as needed
- Consistency
  - Handled by the programmer
- Isolation
  - Concurrent transactions don't interfere
  - Guaranteed through mechanisms such as "locking" and "snapshot isolation"
- Durability
  - Through use of log and replication

Shared Data Structures

- Buffer Pool
  - Manages the disk blocks that are currently being used by the different workers
  - Use some replacement strategy like Least-recently-used
- Log Tail
  - All updates generate “log” records that need to properly numbered and flushed to disk
- Lock Table
  - For synchronization across workers in case of conflicts
- Client Communication Buffers
  - To keep track of what data has already been sent back to clients, and to buffer more outputs

Storage Management

- Databases need to be able to control:
  - Where data is physically stored on the storage devices, especially what is sequentially stored (i.e., spatial locality)
    - To reduce/estimate costs of operations
  - What is in memory vs not in memory (temporal locality)
    - To optimize query execution
  - How is memory managed
    - To avoid double copying of data
  - In which order data is written out of volatile storage (memory) into non-volatile storage (disks/SSDs)
    - For guaranteeing correctness in presence of failures
- Operating systems often get in the way
  - Databases often allocate a large file on disk and manage spatial locality themselves (no guarantees that the file is sequential though)
  - Use memory mapping to reduce double copying within memory
  - And many other tricks to get around OS restrictions…

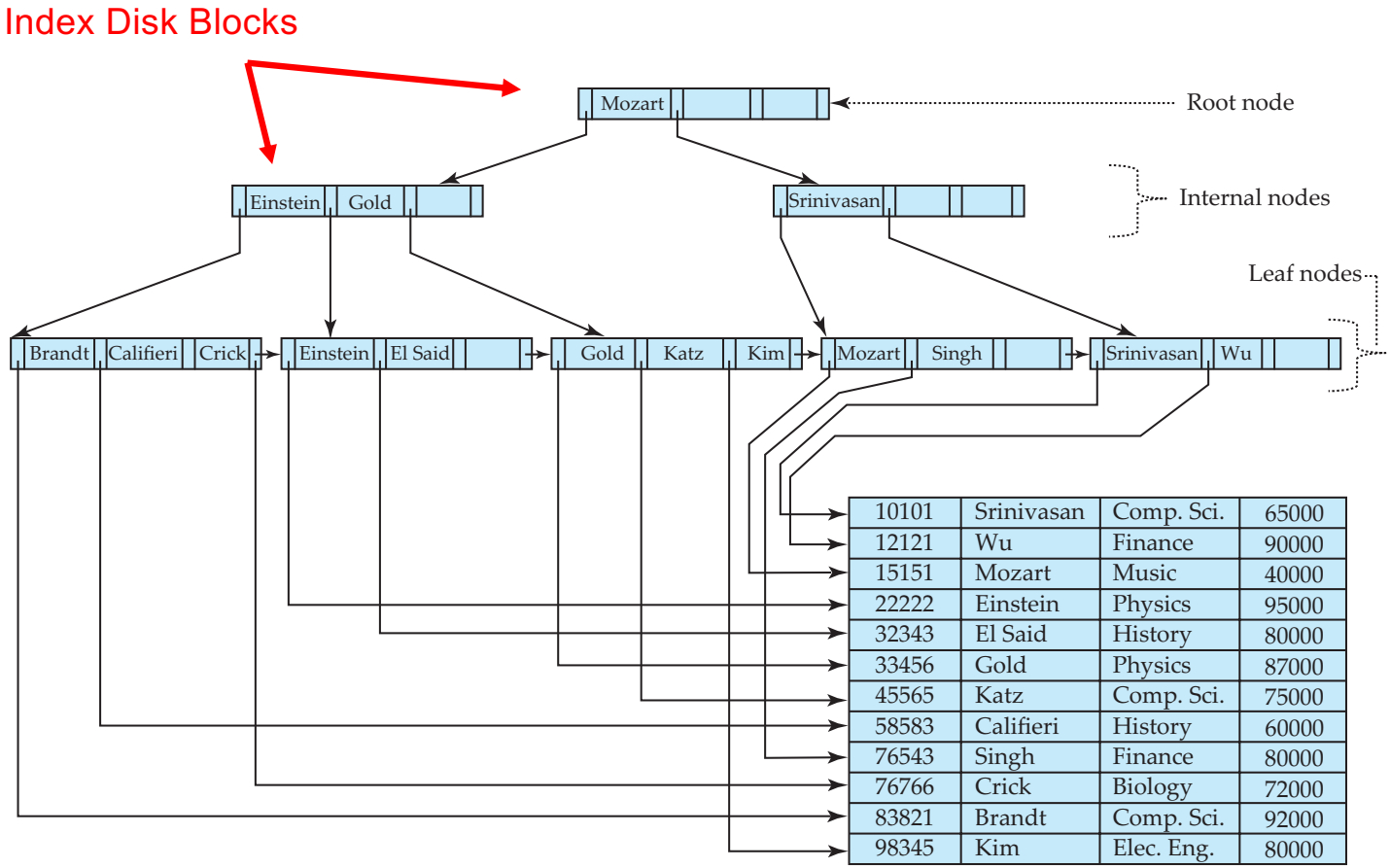
Outline

- Architecture of a Database System
- **AutoAdmin: Self-tuning Databases**
- Towards Self-driving Databases

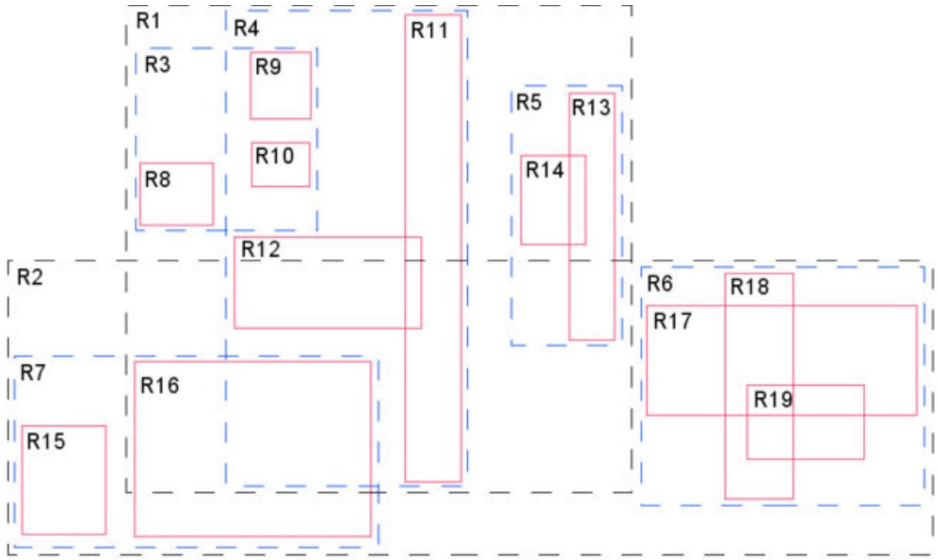
Databases Background: Indexes

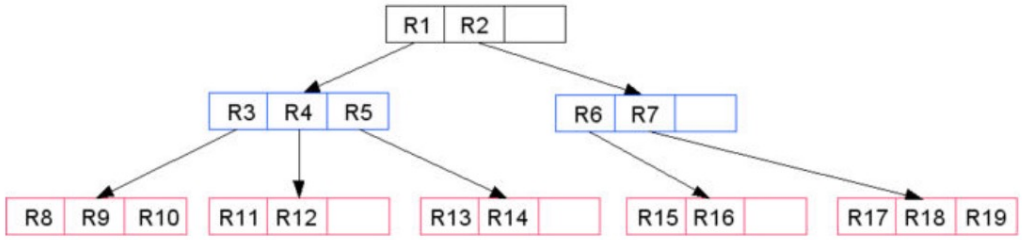
- Data structures built for faster search
  - i.e., queries that want to find tuples with specific properties
  - Search key == the attribute on which you want to search on
- B+-Tree Indexes
  - Primary: data ordered by search key, Secondary: data not ordered by search key
  - Today: "log-structured merge trees" are more common for this usercase
- Hash Indexes (i.e., a persistent hash table)
  - Good for "equality", but doesn't work well for "range" queries (10 < a < 20)
- Multi-dimensional indexes
  - Can support searches on multiple attributes simultaneously
- Spatial indexes
  - Support searches on spatial data (e.g., find all rectangles that overlap with this rectangle)

A Secondary B+-Tree Index



R-Trees (for rectangles)





A Timeline

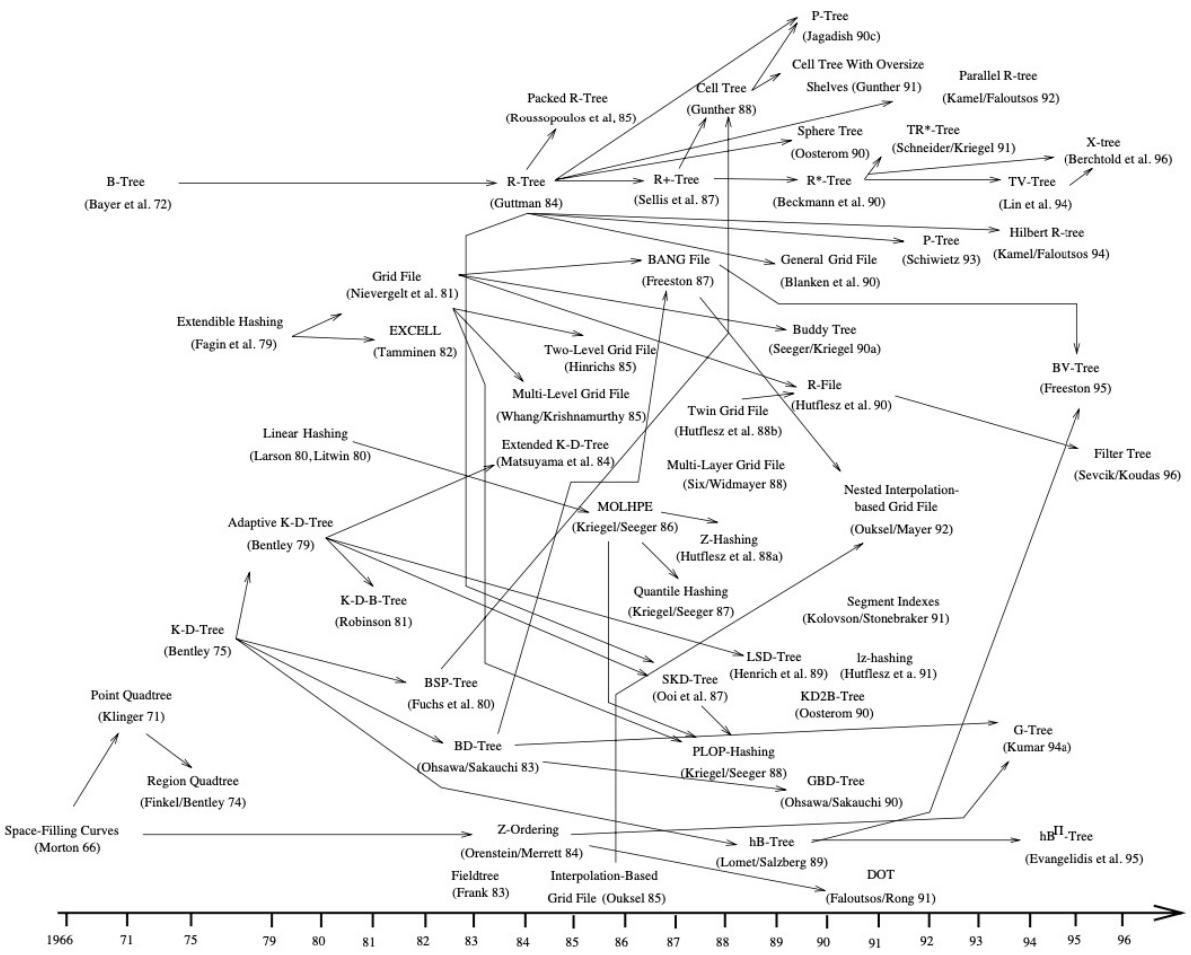


Figure: A Timeline of Indexes (From *Multidimensional Access Methods; Gaede, Gunther; ACM Surveys 1998*)

Databases Background: Indexes

- B+-Tree Indexes traditionally the most common in Relational Databases
  - More suitable for use on hard disks
- Today: log-structured merge trees considered superior
  - Better use of memory and better for SSDs
- Hash Indexes more widely used in distributed systems
- Other types of indexes used primarily in specialized systems (e.g., GIS)

- Drawbacks:
  - Expensive updates
  - Space requirements

Databases Background: Materialized Views

- Originally: "named" expressions

```
create view physics_fall_2009 as
    select course.course_id, sec_id, building, room_number
    from course, section
    where course.course_id = section.course_id
                    and course.dept_name = 'Physics'
                    and section.semester = 'Fall'
                    and section.year = '2009';
```

- Can be used in any query

```
select course_id
from physics_fall_2009
where building= 'Watson';
```

Databases Background: Materialized Views

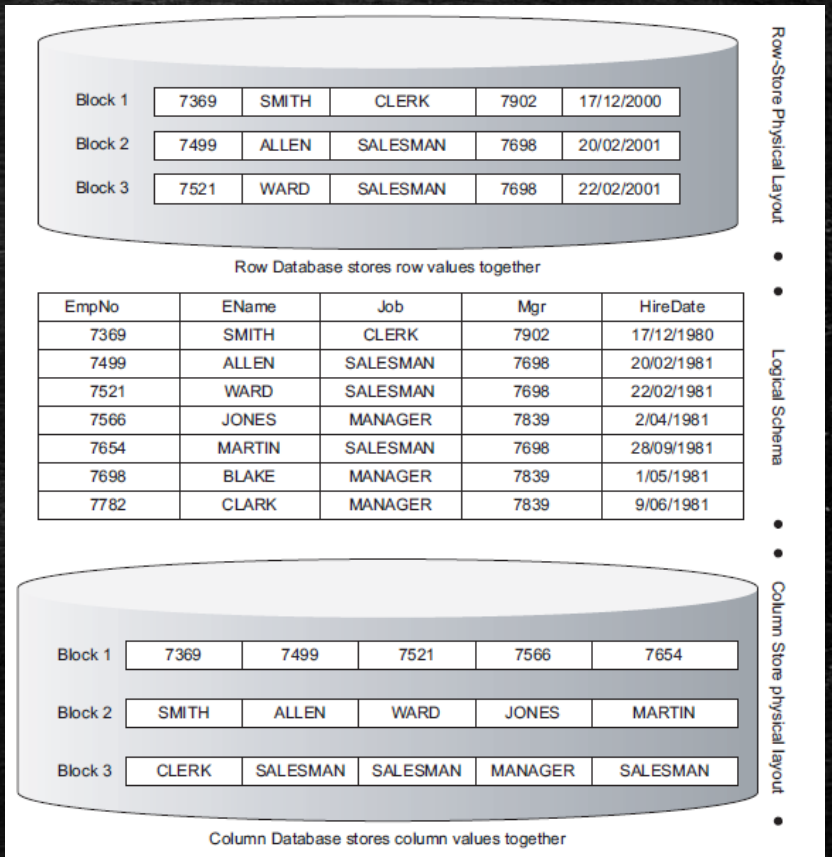
- Can be pre-computed and stored ➔ materialized
- Faster query performance, but must be updated on every insert to the base relations

item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4

- Many other types of summaries common
- Data cubes (multi-dimensional aggregates)
  - Widely used in BI and data viz tools
- Projections
- Indexes can be considered pre-computed structures also

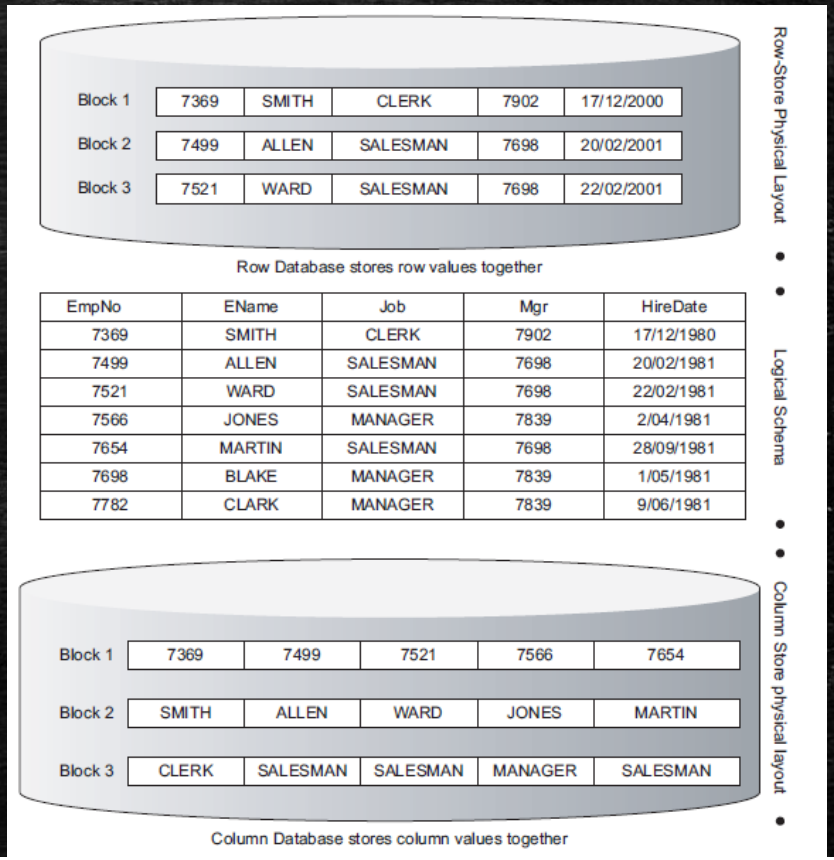
Databases Background: Partitioning

- **Vertical partitioning, aka columnar storage**
- Very common in data warehouses today
  - Also in data lakes ("Parquet" format)
- Speeds up queries substantially
  - Only retrieve data that's needed
- Writes are more expensive
  - Need to update many different blocks/files



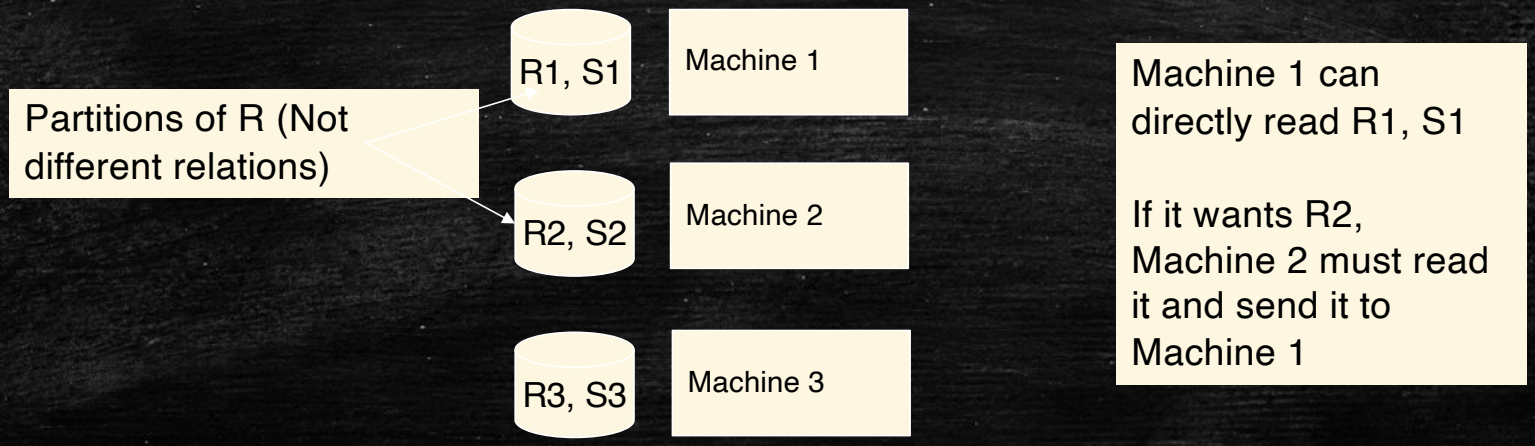
Databases Background: Partitioning

- **Vertical partitioning, aka columnar storage**
- Very common in data warehouses today
  - Also in data lakes ("Parquet" format)
- Speeds up queries substantially
  - Only retrieve data that's needed
- Writes are more expensive
  - Need to update many different blocks/files



Databases Background: Partitioning

- **Horizontal Partitioning, or Sharding**
- Exploit parallelism naturally, for both reads and writes
- Coordination more involved -- require expensive "shuffles" to collect data
- Choice of partitioning attribute/strategy has significant impact



Motivation for AutoAdmin

- Physical data independence → can change physical structures without affecting users
- Older work
  - Stonebraker (1974): Choosing indexes using a parametric model
  - Finkelstein et al. (1988): Collect workload (SQL queries and frequencies) and use to decide which indexes to build
- Big question
  - How to decide whether a proposed "configuration" is good
    - In general: any change to the physical structures/design

Evaluating Configurations

- Option 1: Create a cost model that can assign a "score" to a configuration
  - Too many complex interactions between memory/disk/CPU, and between the operations (updates/queries)
- Option 2: Try it out
  - Infeasible
- Option 3: Ask the "query optimizer", already has a cost model
  - Still need to built the relevant statistics
  - Can be quite expensive

VLDB 1997 Paper

- Challenges
  - Much more complex optimizers
  - Indexes used in many more ways than before
  - Only option: involve the query optimizer
- Key issues
  - How to support this architecturally
  - How to build the relevant statistics (that the optimizer needs) efficiently
  - How to handle the more complex indexes
  - How to do this at large scale, with big workloads

VLDB 1997 Paper

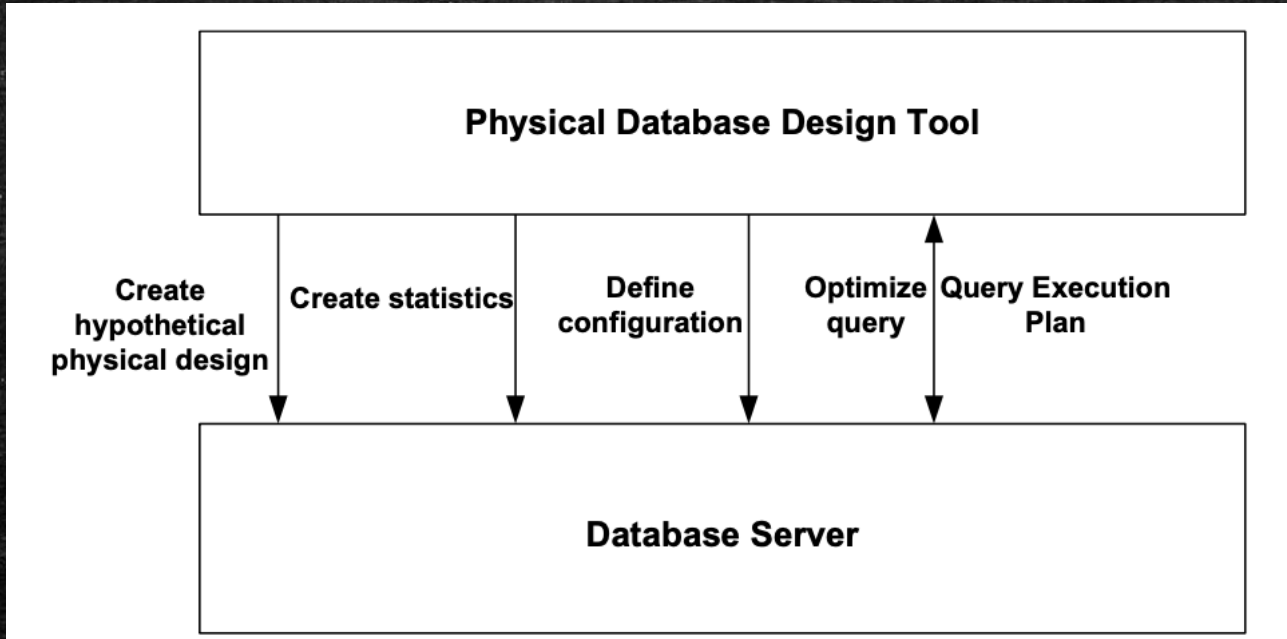


Figure 1. "What-if" analysis architecture for physical database design.

VLDB 1997 Paper

- "Create Hypothetical Index" command to create a proposed index
- "Create Statistics" using sampling to reduce time
- An optimization mode that told the optimizer which indexes to consider
  - Otherwise would need to repeatedly drop/create indexes
  - "Define Configuration" before invoking optimizer
- Heuristics beyond that to restrict the search space of candidate indexes
  - Use approximations while computing costs, prune aggressively
- Shipped as Index Tuning Wizard in MS SQL Server

Materialized Views and Partitioning

- Space of candidates much much larger
- Developed a more general architecture to handle all such decisions
- Many heuristics to prune down the search space and reduce the overall cost of tuning
- Shipped as Database Engine Tuning Advisor in SQL Server 2005

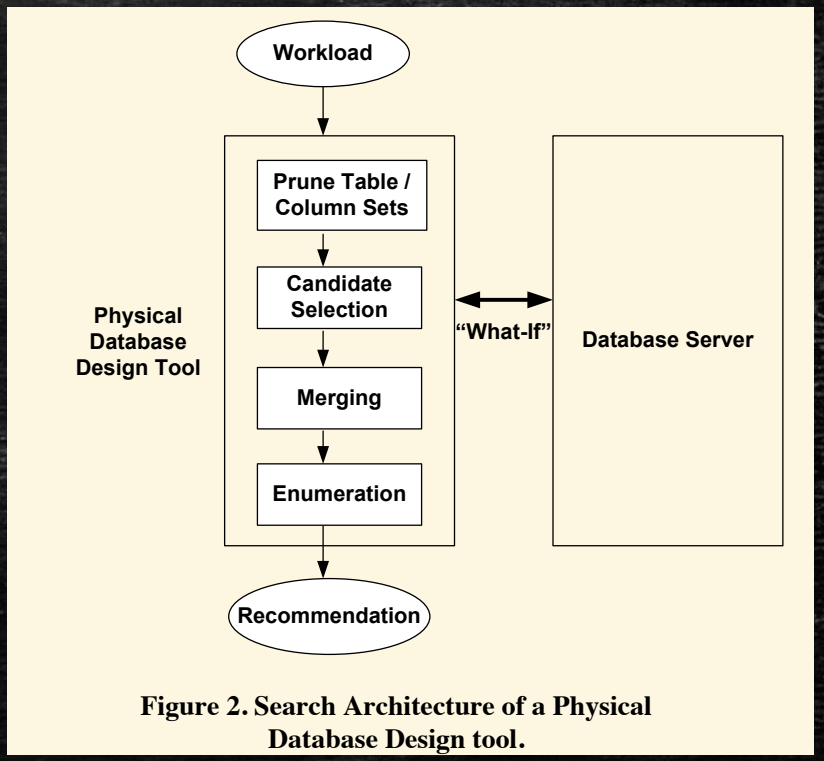


Figure 2. Search Architecture of a Physical Database Design tool.

Follow-up Work

1. Use the optimizer to generate the candidates
   - Instead of trying to guess based on the query structure

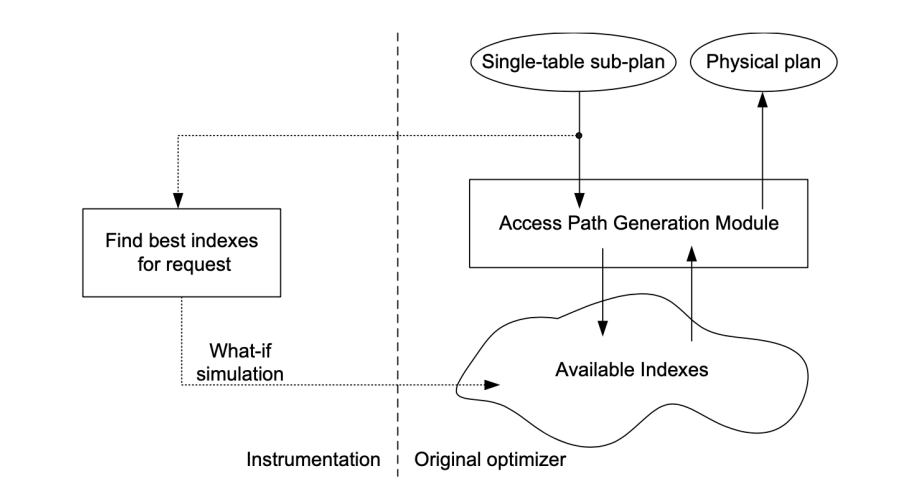


Figure 2: Instrumenting the Query Optimizer.

Follow-up Work

2. Lightweight monitoring tool that alerts the DBA if a significant tuning opportunity exists
3. Consider the "workload" as a sequence and exploit patterns
4. Dynamic, online tuning
   - Need to ensure no back and forth
   - DBAs traditionally wary of such tools

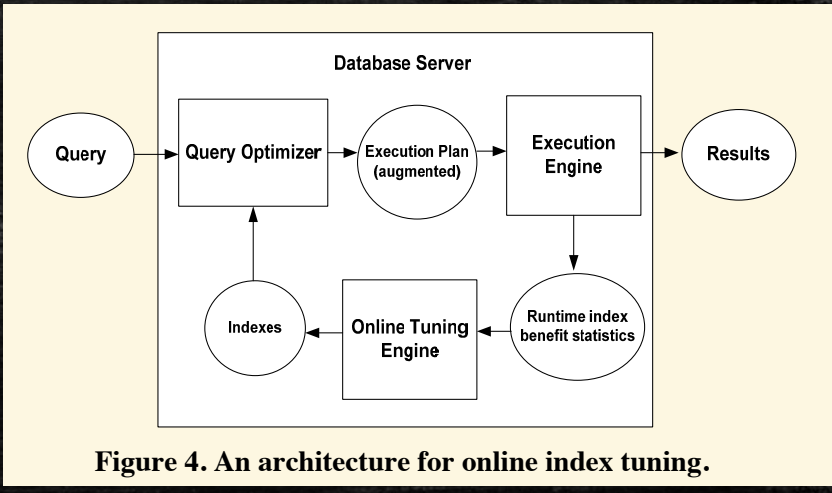


Figure 4. An architecture for online index tuning.

Other Self-Tuning Work

- Statistics and histograms
  - Important to choose the right set for the optimizer to be effective
- Monitoring of Internal Progress or State
  - e.g., query progress
- Learning Optimizer (LEO) in DB2
  - Keeps track of where the optimizer made biggest mistakes
- Adaptive Query Processing
  - Will cover later

Other Self-Tuning Work

- Statistics and histograms
  - Important to choose the right set for the optimizer to be effective
- Monitoring of Internal Progress or State
  - e.g., query progress
- Learning Optimizer (LEO) in DB2
  - Keeps track of where the optimizer made biggest mistakes
- Adaptive Query Processing
  - Will cover later
- GMAP: A more radical approach to physical design

Future Directions

- Benchmarking/comparing different approach for self-tuning
- More lightweight approaches to do tuning
  - e.g., partial indexes/materialized views, small changes to physical layout at a time
- Multi-tenant systems
  - Workload forecasting much harder
- How to use ML, Control Theory, etc.

Some Questions

- How relevant are these types of techniques in modern DMS?
- Can the dependence on the optimizer be reduced?
- How much are we being held back by legacy decisions?
- How do we move towards a truly autonomous, self-driving DMS?
- When and how often should the updates be done?