The Architecture of a DBMS

Based on “Architecture of a database system” by Hellerstein, Stonebraker, and Hamilton (2007)
Process Model

Mapping concurrent client requests to OS processes/threads

- **OS processes**
  - Large state limits #processes

- **Kernel-level Threads**
  - Kernel schedules threads, non-blocking syscalls

- **User-level Threads (Lightweight Threads)**
  - Faster context switching/scheduling, but blocking syscalls

- **DBMS client implements the API to communicate with DBMS**

- **DBMS worker is a thread in DBMS that does work on behalf of a client**
Process models

Process per DBMS worker
- easy to implement, isolates workers from one another
- Requires extensive use of shared memory
- Limits scalability as \#processes is limited
- still heavily used today.

Thread per DBMS worker
- One process with a thread per worker
- Threads can overrun each other, hard to debug and port
Process models

Process pool
- central process holds all DBMS client connections
- Client requests are given to one of the processes in a standby process pool
- More memory efficient
- Has advantages of process per worker and better scalability

Shared data among processes
- Database I/O requests and the Buffer pool
- Log I/O requests and the Log Tail (in-memory log entries waiting to be flushed to disk)
- Lock Table
- Client communication buffers
Increasing workload in any multi-user system beyond a level leads to thrashing often due to memory pressure (working set can not be kept in memory)

- In DBMSs, query processing techniques (eg sorting, hash joins) consume large amounts of memory
- Admission control queues additional work and provides graceful degradation (eg throughput is maintained while latencies increase proportionally to the arrival rate)

Can be done
- at dispatcher by queueing incoming client connections
- at query execution by queueing plans waiting to start execution (taking into account available resources and plan’s estimated resource (CPU+memory) needed)
Sharing and Parallelism

- shared-memory system :: all processors can access the same RAM and disk with roughly the same performance.
- All three process models run well of shared memory systems
- Non-Uniform Memory Access (NUMA) systems
**Shared-nothing parallelism**

- shared-nothing system :: cluster of independent machines that communicate over a high-speed network
  - no way to directly access the memory or disk of another machine
  - run a standard process model on each machine in the cluster
- horizontal data partitioning and explicit cross-processor coordination for transactions, load sharing, etc
- Good fault-tolerance
**Shared-disk parallelism**

- Shared-disk system :: processors can access the disks with about the same performance, but can’t access other’s RAM.
- shared-disk have lower admin costs than shared-nothing system
- Less sensitive to failure of a node
- Shared-disk systems depend upon a distributed lock manager facility, and a *cache-coherency* protocol for managing the distributed buffer pools.

![Diagram of shared-disk parallelism](image)
Query processor

- query processor takes an SQL statement, validates it, optimizes it into a procedural dataflow execution plan, and upon admission executes that plan on behalf of a client

Query parsing and authorization

- Canonicalize table names in the FROM clause to server.database.schema.table.
  - invoke the catalog manager to check that the table is registered, ensure that attribute references are correct.
- Check authorizations (does user have appropriate permissions on the tables, UDFs, etc referenced in the query
  - Some systems perform security checks at execution (to support tuple-level security or share query plans among users)
Query processor

Query rewrite simplifies and normalizes query without changing its semantics

- can rely only on the query and on metadata in the catalog, and cannot access data in the tables
- View expansion
- Constant arithmetic evaluation
- Logical rewriting of predicates in WHERE clause
- Semantic optimization (eg using integrity/FK constraints to simplify joins)
- Nested Subquery flattening and other heuristic rewrites
Query optimizer

transform an internal query representation into an efficient
query plan for execution

- Queries are broken into SELECT-FROM-WHERE query blocks
- Individual query blocks are optimized
  - performance metric of merit block I/Os, time to query completion, time to first row, etc
- Few operators are added to handle GROUP BY, ORDER BY, HAVING, DISTINCT etc
- Blocks are stitched together
- query plan can be represented in machine code or be interpretable
Query executor

- Fully specified query plan = directed dataflow graph that connects operators that encapsulates base table access and various query execution algorithms

- Most modern query executors employ the iterator model (pull or push)
  - API:: init(), get_next(), close()

- Iterators couple dataflow with control flow (control is returned to caller upon returning a tuple)
  - Iterators can be used as input to other iterators
  - All operators in a query plan are subclasses of iterator class
Iterators

- each iterator is pre-allocated some *tuple descriptors*
  - one for each of its inputs, and one for its output.
  - tuple descriptor = array of column references (pointer to a tuple in memory and offset within that tuple)

Where are the tuples in main memory?

- in pages in the buffer pool (BP-tuples); pinning page until reference is cleared
- Allocated memory in the heap by the iterator (M-tuple) and copy data from BP-tuple
Iterators & Updates

- Often execution plans for INSERT, DELETE, UPDATE statements look like straight line query plans.
- Sometimes, things get complicated.
  - UPDATE EMP SET SALARY=SALARY * 1.1 WHERE SALARY < 10
- Recall that an SQL statement is not allowed to “see” its own updates.
- Solution: materialize list of record-ids to be modified, and feed the list to update operator.
Access methods

Access methods = routines that manage access to the various disk-based data structures that the system supports

- Heaps, B+-trees, hash indexes, etc

Access methods support an iterator API with

- init(SearchArgument) method
- get_next() returning with a tuple satisfying the SARG or null

Provide a clean architectural boundary between query and storage engines while supporting excellent performance

access methods have deep interactions with the concurrency and recovery modules
Data warehouses

- Data warehouse=large historical databases for decision-support that are loaded with new data on a periodic basis
- On-Line Transaction Processing (OLTP) vs off-line processing
- Often schema for historical data does not match schema for now data
  - Extract, Transform, and Load (ETL) systems to address these diffs
- Need for fast bulk loads
- Mostly read-only => use Bitmap indexes for enumerated attributes
Many queries join multiple large tables and have a tendency to run “forever”.

Materialized views (selecting the views to materialize, maintaining the freshness of the views, considering the use of materialized views in ad-hoc queries)

most business analytics queries ask for aggregates

data cubes=materialized views of such aggregates

systems that navigate data cubes are known as on-line analytical processing (OLAP)

some claim that column stores have a huge advantage in the data warehouse space vs row stores