Anatomy of a Database System

How is it implemented?

Issues:
  - Process models
  - Parallelism
  - Storage models
  - Buffer manager
  - Query processing architecture
  - Transaction processing
  - Etc...
Introduction

Fig. 1.1 Main components of a DBMS.

A well-understood point of reference for new extensions and revolutions in database systems that may arise in the future. As a result, we focus on relational database systems throughout this paper.

At heart, a typical RDBMS has five main components, as illustrated in Figure 1.1. As an introduction to each of these components and the way they fit together, we step through the life of a query in a database system. This also serves as an overview of the remaining sections of the paper.

Consider a simple but typical database interaction at an airport, in which a gate agent clicks on a form to request the passenger list for a flight. This button click results in a single-query transaction that works roughly as follows:

1. The personal computer at the airport gate (the “client”) calls an API that in turn communicates over a network to establish a connection with the Client Communications Manager of a DBMS (top of Figure 1.1). In some cases, this connection
Processes

- Heavyweight, context switch expensive
- Costly to create, limits on how many
- Large address space, OS support from the beginning
Process Models

- **Processes**
  - Heavyweight, context switch expensive
  - Costly to create, limits on how many
  - Large address space, OS support from the beginning

- **Threads**
  - Lightweight, more complicated to program
  - No OS support till recently
  - In theory, can have very large numbers, in practice, not lightweight enough

Huge implications on performance

Many DBMS wrote their own operating systems, their own thread packages etc...
Process Models

- **Processes**
  - Heavyweight, context switch expensive
  - Costly to create, limits on how many
  - Large address space, OS support from the beginning

- **Threads**
  - Lightweight, more complicated to program
  - No OS support till recently
  - In theory, can have very large numbers, in practice, not lightweight enough

- **Huge implications on performance**
  - Many DBMS wrote their own operating systems, their own thread packages etc...
Process Models

- Assume: Uniprocessors + OS support for efficient threads
- Option 1: “Process per connection”
  - Not scalable (1000 Xion/s?), Shared data structures
  - OS manages time-sharing, easy to implement
Assume: Uniprocessors + OS support for efficient threads

Option 2: “Server Process Model”
- Difficult to port/debug, no OS protection. Requires asynchronous I/O.

Figure: Server Process Model
- Assume: Uniprocessors + OS support for efficient threads
- Option 3: “Server Process + I/O processes”
  - Use I/O processes for handling disks. One process per device.

![Diagram of Process Pool](image.png)

Connections Multiplexed Over Process Pool
DBMS threads, OS processes, OS Threads etc...

- Earlier OSs did not support:
  - Buffering control, asynchronous I/O, high-performance threads
- Many DBMSs implemented their own thread packages
  - Much replication of functionality
DBMS threads, OS processes, OS Threads etc...

- Earlier OSs did not support:
  - Buffering control, asynchronous I/O, high-performance threads
- Many DBMSs implemented their own thread packages
  - Much replication of functionality
- How to map DBMS threads on OS processes/threads?
  - One or more processes/threads to host SQL processing threads
  - One or more “dispatcher processes/threads”
  - One process/thread per disk and one per log disk
  - One coordinator agent process/thread per session
  - Processes/threads for background tools/utilities
Storage Models

- Spatial control
  - Sequential vs random
    - Seeks not improving that fast
  - Controlling spatial locality
    - Directly access to the disk (if possible)
    - Allocate a large file, and address using the offsets
Buffer management

DBMS need control – why?
- Correctness (WAL), performance (read-ahead)
- Typical installations not I/O-bound

Allocate a large memory region
- Maintain a page table with: disk location, dirty bit, replacement policy stats, pin count

Page replacement policy
- LRU-2

“double buffering” issues

Memory-mapping: mmap
**Transactions**

- **Monolithic (why?)**
  - Lock manager, log manager, buffer pool, access methods

- **ACID**
  - Typically:
    - “I” – locking, “D” – logging
    - “A” – locking + logging, “C” – runtime checks
  - **BASE ? (Eric Brewer)**
    - Basically Available Soft-state Eventually consistent
Transactions

- **Locks**
  - Strict 2PL most common
  - Uses a dynamic hash table-based “lock table”
    - Contains: lock mode, holding Xion, waiting Xions etc
    - Also, a way to start the Xion when a lock is obtained

- **Latches**
  - Quick-duration
  - Mostly for internal data structures, internal logic
    - Can’t have deadlocks or other consistency issues
Isolation Levels

- Degrees of consistency (Gray et al.)
  - Read uncommitted, read committed, repeatable read, serializable
  - “Phantom” tuples
  - ANSI SQL Isolation levels
    - Not fully well-defined
Log manager

- Required for atomicity and durability
  - Allows recovery and transaction aborts
  - Why a problem?
    - “STEAL” and “NO FORCE”
  - Concepts:
    - Write-ahead logging, in-order flushes etc
    - Undo/redo, checkpoints
  - ARIES
Locking/Logging and Indexes

- **Locking:**
  - Can’t use 2PL on indexes
  - Solutions: “Crabbing”, Right-link schemes

- **Logging:**
  - No need to “undo” a index page split

- **Phantom problem:**
  1. Use predicate locking
  2. “next-key” locking
Shared Components

- Memory allocations
  - Usually “context”-based
    - Allocate a large context, and do everything within it
  - Why?

- Disk management subsystems
  - Dealing with RAID etc

- Replication services
  - Copy, trigger-based or replay-log

- Statistics gathering, reorganization/index construction, backup/export
Parallelism

(a) shared memory
(b) shared disk
(c) shared nothing
(d) hierarchical
Parallelism

- Shared memory
  - Direct mapping from uni-processor

- Shared nothing
  - Horizontal data partitioning, partial failure
  - Query processing, optimization challenging

- Shared disk
  - Distributed lock managers, cache-coherency etc...
Query Processing

- Assume single-user, single-threaded
  - Concurrency managed by lower layers

Steps:
- Parsing: attribute references, syntax etc...
  - Catalog stored as “denormalized” tables
- Rewriting:
  - Views, constants, logical rewrites (transitive predicates, true/false predicates), semantic (using constraints), subquery flattening
Query Processing

- Steps: Optimizer
  - Block-by-block
  - Machine code vs interpretable
  - Compile-time vs run-time
  - Selinger ++:
    - Larger plan space, selectivity estimation
    - Top-down (SQLServer), auto-tuning, expensive fns

- “Hints”
Steps: Executor

- “get_next()” iterator model
  - Narrow interface between iterators
  - Can be implemented independently
  - Assumes no-blocking-I/O

- Some low-level details
  - Tuple-descriptors
  - Very carefully allocated memory slots
  - “avoid in-memory copies”

- Pin and unpin
Query Processing

- **SQL Update/Delete**
  - “Halloween” problem

- **Access Methods**
  - B+-Tree and heap files
    - Multi-dimensional indexes not common
  - init(SARG)
    - “avoid too many back-and-forth function calls”
  - Allow access by RID