Anatomy of a Database System

- How is it implemented?
- Issues:
  - Process models
  - Parallelism
  - Storage models
  - Buffer manager
  - Query processing architecture
  - Transaction processing
  - Etc...

Overview

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...
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 establishes a process per connection model (Figure 2.1) was used by early DBMS implementations and is still used by many commercial systems today. This model is relatively easy to implement since DBMS workers are mapped directly onto OS processes. The OS scheduler manages the timesharing of DBMS workers and the DBMS programmer can rely on OS protection facilities to isolate standard bugs like memory overruns. Moreover, various programming tools like debuggers and memory checkers are well-suited to this process model. Complicating this model are the in-memory data structures that are shared across DBMS connections, including the lock table and buffer pool (discussed in more detail in Sections 6.3 and 5.3, respectively). These shared data structures must be explicitly allocated in OS-supported shared memory accessible across all DBMS processes. This requires OS support (which is widely available) and some special DBMS coding. In practice, the

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

Figure 1: Process per Connection
Process Models

- Assume: Uniprocessors + OS support for efficient threads
- Option 2: “Server Process Model”
  - Difficult to port/debug, no OS protection. Requires asynchronous I/O.

![Server Process Model](image)

Figure 2: Server Process Model

Process Models

- 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.

Process Models

- 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
Process Models

Fig. 2.3 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 processed.

and all processes are already servicing other requests, the new request must wait for a process to become available.

Process pool has all of the advantages of process per DBMS worker but, since a much smaller number of processes are required, is considerably more memory efficient.

Process pool is often implemented with a dynamically resizable process pool where the pool grows potentially to some maximum number when a large number of concurrent requests arrive. When the request load is lighter, the process pool can be reduced to fewer waiting processes. As with thread per DBMS worker, the process pool model is also supported by a several current generation DBMS in use today.

2.1.4 Shared Data and Process Boundaries

All models described above aim to execute concurrent client requests as independently as possible. Yet, full DBMS worker independence and isolation is not possible, since they are operating on the same shared

Figure 3: Server Process + I/O Processes

Parallelism

![Parallelism Diagram](https://via.placeholder.com/150)

(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...
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

Storage Models

- 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

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”

Query Processing

• 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
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” an 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