Database Implementation For Modern Hardware

Thomas Neumann
Introduction

Database Management Systems (DBMS) are extremely important
- used in nearly all commercial data management
- very large data sets
- valuable data

Key challenges:
- scalability to huge data sets
- reliability
- concurrency

Results in very complex software.
About This Lecture

Goals of this lecture

- learning how to build a modern DBMS
- understanding the internals of existing DBMSs
- understanding the effects of hardware behavior

Rough structure of the lecture

1. the classical DBMS architecture
2. efficient query processing
3. adapting the architecture to hardware trends
Literature


Unfortunately mainly cover the classical architecture.
Motivational Example

Why is a DBMS different from most other programs?

- many difficult requirements (reliability etc.)
- but a key challenge is **scalability**

Motivational example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Looks simple...
Motivational Example (2)

*Given two lists* $L_1$ *and* $L_2$, *find all entries that occur on both lists.*

Simple if both fit in main memory
Motivational Example (2)

Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.

Simple if both fit in main memory

- sort both lists and intersect
- or put one in a hash table and probe
- or build index structures
- or ...

Note: pairwise comparison is not an option! $O(n^2)$
Motivational Example (3)

Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.

Slightly more complex if only one fits in main memory
Motivational Example (3)

*Given two lists* $L_1$ *and* $L_2$, *find all entries that occur on both lists.*

Slightly more complex if only one fits in main memory

- load the smaller list into memory
- build index structure/sort/hash/...
- scan the larger list
- search for matches in main memory

Code still similar to the pure main-memory case.
Motivational Example (4)

Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.

Difficult if neither list fits into main memory
Motivational Example (4)

Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.

Difficult if neither list fits into main memory

- no direct interaction possible
- sorting works, but already a difficult problem
- or use some kind of partitioning scheme
- breaks the problem into smaller problem
- until main memory size is reached

Code significantly more involved.
Motivational Example (5)

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Hard if we make no assumptions about $L_1$ and $L_2$. 
Motivational Example (5)

Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.

Hard if we make no assumptions about $L_1$ and $L_2$.

- tons of corner cases
- a list can contain duplicates
- a single duplicate might exceed main memory!
- breaks “simple” external memory logic
- multiple ways to solve this
- but all of them somewhat involved
- and a DBMS must not make assumptions about its data!

Code complexity is very high.
Motivational Example (6)

Designing scalable algorithm is a hard problem

- must cope with very large instances
- hard even in main memory
- billions of data items
- rules out any $O(n^2)$ algorithm
- external algorithms are even harder

This requires a careful software architecture.
Set-Oriented Processing

Small applications often loop over their data

- one for loop accesses all item $x$,
- for each item, another loop access item $y$,
- then both items are combined.

This kind of code of code feels “natural”, but is bad

- $\Omega(n^2)$ runtime
- does not scale

Instead: set oriented processing. Perform operations for large batches of data.
Relational Algebra

Notation:
- \( \mathcal{A}(e) \) attributes of the tuples produces by \( e \)
- \( \mathcal{F}(e) \) free variables of the expression \( e \)
- binary operators \( e_1 \theta e_2 \) usually require \( \mathcal{A}(e_1) = \mathcal{A}(e_2) \)

\[ e_1 \cup e_2 \quad \text{union, } \{ x \mid x \in e_1 \lor x \in e_2 \} \]
\[ e_1 \cap e_2 \quad \text{intersection, } \{ x \mid x \in e_1 \land x \in e_2 \} \]
\[ e_1 \setminus e_2 \quad \text{difference, } \{ x \mid x \in e_1 \land x \notin e_2 \} \]
\[ \rho_{a \to b}(e) \quad \text{rename, } \{ x \circ (b : x.a) \setminus (a : x.a) \mid x \in e \} \]
\[ \Pi_A(e) \quad \text{projection, } \{ \circ_{a \in A}(a : x.a) \mid x \in e \} \]
\[ e_1 \times e_2 \quad \text{product, } \{ x \circ y \mid x \in e_1 \land y \in e_2 \} \]
\[ \sigma_p(e) \quad \text{selection, } \{ x \mid x \in e \land p(x) \} \]
\[ e_1 \bowtie_p e_2 \quad \text{join, } \{ x \circ y \mid x \in e_1 \land y \in e_2 \land p(x \circ y) \} \]

per definition set oriented. Similar operators also used bag oriented (no implicit duplicate removal).
Relational Algebra - Derived Operators

Additional (derived) operators are often useful:

- **Natural Join**: $e_1 \bowtie e_2$
  \[ \{ x \circ y | A(e_2) \setminus A(e_1) | x \in e_1 \land y \in e_2 \land x = A(e_1) \cap A(e_2) \land y \} \]

- **Division**: $e_1 \div e_2$
  \[ \{ x | A(e_1) \setminus A(e_2) | x \in e_1 \land \forall y \in e_2 \exists z \in e_1 : y = A(e_2) z \land x = A(e_1) \cap A(e_2) z \} \]

- **Semi-Join**: $e_1 \bowtie_p e_2$
  \[ \{ x | x \in e_1 \land \exists y \in e_2 : p(x \circ y) \} \]

- **Anti-Join**: $e_1 \triangleright_p e_2$
  \[ \{ x | x \in e_1 \land \forall y \in e_2 : p(x \circ y) \} \]

- **Outer Join**: $(e_1 \bowtie_p e_2) \cup \{ x \circ a \in A(e_2) | a : null \} | x \in (e_1 \triangleright_p e_2) \}$

- **Full Outer Join**: $(e_1 \bowtie_p e_2) \cup (e_2 \bowtie_p e_1)$
Relational Algebra - Extensions

The algebra needs some extensions for real queries:

- map/function evaluation
  \[ \chi_{a:f}(e) = \{ x \circ (a : f(x)) | x \in e \} \]

- group by/aggregation
  \[ \Gamma_{A:a:f}(e) = \{ x \circ (a : f(y)) | x \in \Pi_A(e) \land y = \{ z | z \in e \land \forall a \in A : x.a = z.a \} \} \]

- dependent join (djoin). Requires \( \mathcal{F}(e_2) \subseteq A(e_1) \)
  \[ e_1 \bowtie_p e_2 = \{ x \circ y | x \in e_1 \land y \in e_2(x) \land p(x \circ y) \} \]
Set-Oriented Processing (2)

Processing whole batches of tuples is more efficient:

- can prepare index structures
- or re-organize the data
- sorting/hashing
- runtime ideally $O(n \log n)$

Many different algorithms, we will look at them later.
Traditional Assumptions

Historically, DBMS are designed for the following scenario:

- data is much larger than main memory
- I/O costs dominate everything
- random I/O is very expensive

This led to a very conservative, but also very scalable design.
Hardware Trends

Hardware development changed some of the assumptions

- main memory size is increasing
- servers with 1TB main memory are affordable
- flash storage reduces random I/O costs
- ...

This has consequences for DBMS design

- CPU costs become more important
- often I/O is eliminated or greatly reduced
- the old architecture becomes suboptimal

But this is more evolution than revolution. Many of the old techniques are still required for scalability reasons.
Introduction

Goals

Ideally, a DBMS

- handles arbitrarily large data sets efficiently
- never loses data
- offers a high-level API to manipulate and retrieve data
- shields the application from the complexity of data management
- offers excellent performance for all kinds of queries and all kinds of data

This is a very ambitious goal!
In many cases indeed reached, but implies complexity.
Overview

1. The Classical Architecture
   1.1 storage
   1.2 access paths
   1.3 transactions and recovery

2. Efficient Query Processing
   2.1 set oriented query processing
   2.2 algebraic operators
   2.3 code generation

3. Designing a DBMS for Modern Hardware
   3.1 re-designing storage
   3.2 optimizing cache locality
   3.3 main memory databases
Storage
The Problem
Requirements

There are different classes of requirements:

- **Data Independence**
  - application is shielded from physical storage
  - physical storage can be reorganized
  - hardware can be changed

- **Scalability**
  - must scale to (nearly) arbitrary data sizes
  - fast retrieval
  - efficient access to individual data items
  - updating arbitrary data

- **Reliability**
  - data must never be lost
  - must cope with hardware and software failures

- ...
Layer Architecture

- implementing all these requirements on the “bare metal” is hard
- and not desirable
- a DBMS must be maintainable and extensible

Instead: use a layer architecture

- the DBMS logic is split into levels of functionality
- each level is implemented by a specific layer
- each layer interacts only with the next lower layer
- simplifies and modularizes the code
A Simple Layer Architecture

**Purpose**
- query translation and optimization
- managing records and access paths
- DB buffer and hardware interface

**Access Granularity**
- declarative queries
- sets of records
- records
- page
A Simple Layer Architecture (2)

- layers can be characterized by the data items they manipulate
- lower layer offers functionality for the next higher level
- keeps the complexity of individual layers reasonable
- rough structure: physical $\rightarrow$ low level $\rightarrow$ high level

This is a reasonable architecture, but simplified.
A more detailed architecture is needed for a complete DBMS.
A More Detailed Architecture

- **Device Interface**
  - Read block k, write block k
  - Granularity: block, file
  - Storage allocation: block, file
  - Free space inventory: extent table, page
  - Physical record: track, cylinder...

- **File Interface**
  - Read block k, write block k
  - Granularity: block, file
  - Storage allocation: block, file
  - Free space inventory: extent table...
  - Logical record: physical record...

- **Record Access**
  - Write record, insert in B-tree...
  - Granularity: physical record...
  - Physical data: page structure
  - Physical record: page, segment

- **Record Interface**
  - Find NEXT record, STORE record
  - Granularity: logical record, key...
  - Access paths: physical schema...
  - Page structure: page, segment

- **Query Interface**
  - SQL, ...
  - Granularity: relation, view, ...
  - Logical data: logical schema, integrity constraints
  - Logical record: physical record...

- **DB Buffer**
  - Access page j, release page j
  - Granularity: page, segment
  - Page structure: page table, block map...
  - Logical data: page indexes...

- **Storage**
  - DB
  - Application: relation, view...
  - Logical data: logical record, key...
  - Access paths: access path, physical schema...

- **The Classical Architecture**
  - Storage: free space inventory, extent table...
A More Detailed Architecture (2)

A few pieces are still missing:
- transaction isolation
- recovery

but otherwise it is a reasonable architecture.

Some system deviate slightly from this classical architecture
- most DBMSs nowadays delegate drive access to the OS
- some DBMSs delegate buffer management to the OS (tricky, though)
- a few DBMSs allow for direct logical record access
- ...

Influence of Hardware

Must take hardware into account when designing a storage system.

For a long time dominated by Moore’s Law:

*The number of transistors on a chip doubles every 18 month.*

Indirectly drove a number of other parameters:

- main memory size
- CPU speed
  - no longer true!
- HD capacity
  - start getting problematic, too. density is very high
  - only capacity, not access time

Later we will look at these again.
Memory Hierarchy

- **Register**
- **Cache**
- **Main Memory**
- **External Storage (Online)**
- **Archive Storage (Nearline)**
- **Archive Storage (Offline)**

**Capacity**: 
- Bytes: 1ns
- K-M bytes: <10ns
- G bytes: <100ns
- T bytes: ms
- T bytes: sec
- T-P bytes: sec-min
Memory Hierarchy (2)

There are huge gaps between hierarchy levels
- traditionally, main memory vs. disk is most important
- but memory vs. cache etc. also relevant

The DBMS must aim to maximize locality.
Hard Disk Access

Hard Disks are still the dominant external storage:

- rotating platters, mechanical effects
- transfer rate: ca. 150MB/s
- seek time ca. 3ms
- huge imbalance in random vs. sequential I/O!

The DBMS must take these effects into account

- sequential access is much more efficient
- gap is growing instead of shrinking
- even SSDs are slightly asymmetric (and have other problems)
Hard Disk Access (2)

Techniques to speed up disk access:

- do not move the head for every single tuple
- instead, load larger chunks
- typical granularity: one page
- page size varies. traditionally 4KB, nowadays often 16K and more
- page size is a trade-off
Hard Disk Access (3)

The page structure is very prominent within the DBMS
- granularity of I/O
- granularity of buffering/memory management
- granularity of recovery

Page is still too small to hide random I/O though
- sequential page access is important
- DBMSs use read-ahead techniques
- asynchronous write-back
Buffer Management

Some pages are accessed very frequently
- reduce I/O by buffering/caching
- buffer manager keeps active pages in memory
- limited size, discards/write back when needed
- coupled with recovery, in particular logging

Basic interface:
1. FIX(pageNo,shared)
2. UNFIX(pageNo,dirty)

Pages can only be accessed (or modified) when they are fixed.
Buffer Management

The buffer manager itself is protected by one or more latches.
Buffer Frame

Maintains the state of a certain page within the buffer

- **pageNo**: the page number
- **latch**: a read/writer lock to protect the page
  (note: must not block unrelated pages!)
- **LSN**: LSN of the last change, for recovery
  (buffer manager must force the log before writing)
- **state**: clean/dirty/newly created etc.
- **data**: the actual data contained on the page

(Will usually contain extra information for buffer replacement)

Usually kept in a hash table.
Buffer Replacement

When memory is full, some buffer pages have to be replaced

- clean pages can be simply discarded
- dirty pages have to be written back first
- discarded pages are replaced with new pages
Buffer Replacement - FIFO

First In - First Out

- simple replacement strategy
- buffer frames are kept in a linked list
- inserted at the end, remove from the head
- “old” pages are removed first

Does not take locality into account.
Buffer Replacement - LRU

Least Recently Used

- similar to FIFO, buffer frames are kept in a double-linked list
- remove from the head
- when a frame is unfixed, move it to the end of the list
- “hot” pages are kept in the buffer

A very popular strategy. Latching requires some care.
Buffer Replacement - LFU

Least Frequently Used
- remembers the number of access per page
- in-frequently used pages are remove first
- maintains a priority queue of pages

Sounds plausible, but too expensive in practice.
Buffer Replacement - Second Chance

LRU is nice, but the LRU list is a hot spot.

Idea: use a simpler mechanism to simulate LRU

- one bit per page
- bit is set when page is unfixed
- when replacing, replace pages with unset bit
- set bits are cleared during the process
- strategy is called “second chance” or “clock”

Easy to implement, but a bit crude.
Buffer Replacement - 2Q

Maintain not one queue but two

- many pages are referenced only once
- some pages are hot and reference frequently
- maintain a separate list for those

1. maintain all pages in FIFO queue
2. when a page is references again that is currently in FIFO, move it into an LRU queue
3. prefer evicting from FIFO

Hot pages are in LRU, read-once pages in FIFO. Good strategy for common DBMS operations.
Buffer Replacement - Hints

Application knowledge can help buffer replacement

- 2Q tries to recognize read-once pages
- these occur when scanning over data
- but the DBMS knows this anyway!
- it could therefore give **hints** when unfixing
- e.g., *will-need*, or *will-not-need* (changes placement in queue)
Segments

While page granularity is fine for I/O, it is somewhat unwieldy

- most structures within a DBMS span multiple pages
- relations, indexes, free space management, etc.
- convenient to treat these as one entity
- all DBMS pages are partitioned into sets of pages

Such a set of pages is called a segment.

Conceptually similar to a file or virtual memory.
Segments (2)

A segment offers a virtual address space within the DBMS

- can allocate and release new pages
- can iterate over all pages
- can drop the whole segment
- optionally offers a linear address space

Greatly simplifies the logic of higher layers.
Block Allocation

- **static file-mapping**
- **dynamic extent-mapping**
- **dynamic block-mapping**
Block Allocation

All approaches have pros and cons:

- **static file-mapping**
  - very simple, low overhead
  - resizing is difficult

- **dynamic block-mapping**
  - maximum flexibility
  - administrative overhead, additional indirection

- **dynamic extent-mapping**
  - can handle growth
  - slight overhead

In most cases extent-based mapping is preferable.
Block Allocation (5)

Dynamic extent-mapping:
- grows by adding a new extent
- should grow exponentially (e.g., factor 1.25)
- bounds the number of extents
- reduces both complexity and space consumption
- and helps with sequential I/O!
Segment Types

Segments can be classified into types

- private vs. public
- permanent vs. temporary
- automatic vs. manual
- with recovery vs. without recovery

Differ in complexity and required effort.
Standard Segments
Most DBMS will need at least two low-level segments:

- segment inventory
  - keeps track of all pages allocated to segments
  - keeps extent lists or page tables or ...

- free space inventory
  - keeps track of free pages
  - maintains bitmaps or free extents or ...

High-level segments built upon these.

Common high-level segments:

- schema
- relations
- temp segments (created and discard on demand)
- ...
Update Strategies

DBMSs have different update behavior

<table>
<thead>
<tr>
<th></th>
<th>steal</th>
<th>¬ steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>force</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬ force</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- usually one prefers steal, ¬ force
- but then pages contain dirty data
- when using *update-in-place*, dirty data is on disk
- complicates recovery
- transactions could see dirty data
Shadow Paging

uses a page table, dirty pages are stored in a shadow copy.
Shadow Paging (2)

Advantages:
- the clean data is always available on disk
- greatly simplified recovery
- can be used for transaction isolation, too

Disadvantages:
- complicates the page access logic
- destroys data locality

Nowadays rarely used in disk-based systems.
Delta Files

Similar idea to shadow paging:

- on change pages are copied to a separate file
- a copied page can be changed in-place
- on commit discard the file, on abort copy back

Can be implemented in two flavors:

- store a clean copy in the delta
- store the dirty data in the delta

Both have pros and cons.
Delta Files (2)

Delta files have some advantages over shadow paging:

- preserve data locality
- no mixture of clean and dirty pages

Disadvantages:

- cause more I/O
- abort (or commit) becomes expensive
- keeping track of delta pages is non-trivial

Still, often preferable over shadow paging.
Access Paths
Data Structures

The DBMS needs several separate data structures

- for the free space management
- for the data itself (storage and retrieval)
- for unusually large data
- for index structures to speed up access

We will look at these in more detail.
Free Space Inventory

Problem: Where do we have space for incoming data?

Traditional solution: free space bitmap

Each nibble indicates the fill status of a given page.
Free Space Inventory (2)

Encode the fill status in 4 bits (some system use only 1 or 2):

- must approximate the status
- one possibility: data size / \( \frac{\text{page size}}{2^{\text{bits}}} \)
- loss of accuracy in the lower range
- logarithmic scale is often better
- \( \lceil \log_2(\text{text size}) \rceil \)
- or a combination (logarithmic for lower range, linear for upper range)

Encodes the free space (alternative: the used space) in a few bits.
Free Space Inventory (3)

When inserting data,
- compute the required FSI entry (e.g., $\leq 7$)
- scan the FSI for a matching entry
- insert the data on this page

Problem:
- linear effort
- FSI is small, for 16KB pages 1 FSI page covers 512MB
  - but scan still not free
  - only 16 FSI values, cache the next matching page (range)
- most pages will be static (and full anyway)
- segments will mostly grow at the end
- cache avoids scanning most of the FSI entries
Allocation

Allocating pages (or parts of a page) benefits from application knowledge
- often larger pieces are inserted soon after each other
- e.g. a set of tuples
- or one very large data item
- should be allocated close to each other

Allocation interface is usually

\[ \text{allocate}(\text{min}, \text{max}) \]

- \( \text{max} \) is a hint to improve data layout
- some interfaces (e.g., segment growth) even implement over-allocation
- reduces fragmentation
Slotted Pages

(TID size varies, but will most likely be at least 8 bytes on modern systems)
Tuples are stored in slotted pages

- data grows from one side, slots from the other
- the page is full when both meet
- updates/deletes complicate issues, though
- might require garbage collection/compactification
Slotted Pages (3)

Header:

- LSN: for recovery
- slotCount: number of used slots
- firstFreeSlot: to speed up locating free slots
- dataStart: lower end of the data
- freeSpace: space that would be available after compactification

Note: a slotted page can contain hundreds of entries!
Requires some care to get good performance.
Slotted Pages (4)

Slot:

- offset: start of the data item
- length: length of the data item

Special cases:
- free slot: offset = 0, length = 0
- zero-length data item: offset > 0, length = 0
Problem:

1. transaction $T_1$ updates data item $i_1$ on page $P_1$ to a very small size (or deletes $i_1$)
2. transaction $T_2$ inserts a new item $i_2$ on page $P_1$, filling $P_1$ up
3. transaction $T_2$ commits
4. transaction $T_1$ aborts (or $T_3$ updates $i_1$ again to a larger size)

TID concept $\Rightarrow$ create an indirection

**but** where to put it? Would have to move $i_1$ and $i_2$. 

---

The Classical Architecture

Access
Slotted Pages (6)

Logic is much simpler if we can store the TID inside the slot
- borrow a bit from the TID (or have some other way to detect invalid TIDs)
- if the slot contains a valid TID, the entry is redirected
- otherwise, it is a regular slot

Depending on page size size, this wastes a bit space.
But greatly simplifies the slotted page implementation.
Slotted Pages (7)

One possible slot implementation:

\[
\begin{array}{cccccc}
T & S & O & O & O & L & L & L \\
\end{array}
\]

1. if \( T \neq 1111111_2 \), the slot points to another record
2. otherwise the record is on the current page
   2.1 if \( S = 0 \), the item is at offset \( O \), with length \( L \)
   2.2 otherwise, the item was moved from another page
      ▶ it is also placed at offset \( O \), with length \( L \)
      ▶ but the first 8 bytes contain the original TID

The original TID is important for scanning.
Record Layout

The tuples have to be materialized somehow.

One possibility: serialize the attributes

| integer | integer | length | string | integer | length | string |

Problem: accessing an attribute is $O(n)$ in worst case.
Record Layout (2)

It is better to store offset instead of lengths

- splits tuple into two parts
- fixed size header and variable size tail
- header contains pointers into the tail
- allows for accessing any attribute in $O(1)$
Record Layout (3)

For performance reasons one should even reorder the attributes

- split strings into length and data
- re-order attributes by decreasing alignment
- place variable-length data at the end
- variable length has alignment 1

Gives better performance without wasting any space on padding.
NULL Values

What about NULL values?
- represent an unknown/unspecified value
- is a special value outside the regular domain

Multiple ways to store it
- either pick an invalid value (not always possible)
- or use a separate NULL bit

NULL bits allow for omitting NULL values from the tuple
- complicates the access logic
- but saves space
- useful if NULL values are common.
Compression

Some DBMS apply compression techniques to the tuples

- most of the time, compression is **not** added to save space!
- disk is cheap after all
- compression is used to **improve performance**!
- reducing the size reduces the bandwidth consumption

Some people really care about space consumption, of course. But outside embedded DBMSs it is usually an afterthought.
Compression (2)

What to compress?
- the larger data compressed chunk, the better the compression
- but: DBMS has to handle updates
- usually rules out page-wise compression
- individual tuples can be compressed more easily

How to compress?
- general purpose compression like LZ77 too expensive
- compression is about performance, after all
- most system use special-purpose compression
- byte-wise to keep performance reasonable
**Compression (3)**

A useful technique for integer: variable length encoding

<table>
<thead>
<tr>
<th>length (2 bits)</th>
<th>data (0-4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1 byte value</td>
</tr>
<tr>
<td>01</td>
<td>2 bytes value</td>
</tr>
<tr>
<td>10</td>
<td>3 bytes value</td>
</tr>
<tr>
<td>11</td>
<td>4 bytes value</td>
</tr>
</tbody>
</table>

Variant A

<table>
<thead>
<tr>
<th>00</th>
<th>1 byte value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>2 bytes value</td>
</tr>
<tr>
<td>10</td>
<td>3 bytes value</td>
</tr>
<tr>
<td>11</td>
<td>4 bytes value</td>
</tr>
</tbody>
</table>

Variant B

<table>
<thead>
<tr>
<th>NULL, 0 bytes value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte value</td>
</tr>
<tr>
<td>2 bytes value</td>
</tr>
<tr>
<td>4 bytes value</td>
</tr>
</tbody>
</table>
Compression (4)

The length is fixed length, the compressed data is variable length

```
fixed fixed len1 len2 len3 len4 comp1 comp2 comp4
```

Problem: locating compressed attributes
- depends on preceding compression
- would require decompressing all previous entries
- not too bad, but can be sped up
- use a lookup tuples per length byte
Compression (5)

Another popular technique: dictionary compression

Dictionary:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Berlin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>München</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Passauerstraße</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuples:

<table>
<thead>
<tr>
<th>city</th>
<th>street</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- stores strings in a dictionary
- stores only the string id in the tuple
- factors out common strings
- can greatly reduce the data size
- can be combined with integer compression
## Compression (6)

Compressing the strings: FSST

<table>
<thead>
<tr>
<th>corpus (uncompressed)</th>
<th>symbol table</th>
<th>corpus (compressed)</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://in.tum.de">http://in.tum.de</a></td>
<td>0 http://</td>
<td>063</td>
</tr>
<tr>
<td><a href="http://cwi.nl">http://cwi.nl</a></td>
<td>1 www.</td>
<td>07</td>
</tr>
<tr>
<td><a href="http://www.uni-jena.de">www.uni-jena.de</a></td>
<td>2 uni-jena</td>
<td>123</td>
</tr>
<tr>
<td><a href="http://www.wikipedia.org">www.wikipedia.org</a></td>
<td>3 .de</td>
<td>1854</td>
</tr>
<tr>
<td><a href="http://www.vldb.org">http://www.vldb.org</a></td>
<td>4 .org</td>
<td>0194</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6 in.tum</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7 cwi.nl</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8 wikipedi</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9 vldb</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>symbol</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>0194</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>

- compresses even short strings
- allow for random access within compressed data
- very fast decompression
Long Records

Data is organized in pages
- many reasons for this, including recovery, buffer management, etc.
- a tuple must fit on a single page
- limits the maximum size of a tuple

What about large tuples?
- sometimes the user wants to store something large
- e.g., embed a document
- SQL supports this via BLOB/CLOB

Requires some mechanism so handle these large records.
Long Records (2)

Simply spanning pages is not a good idea:
- must read an unbounded number of pages to access a tuple
- greatly complicates buffering
- a tuple might not even fit into main memory!
- updates that change the size are complicated
- intermediate results during query processing

Instead, keep the main tuple size down
- BLOBS/CLOBS are stored separate from the tuple
- tuple only contains a pointer
- increases the costs of accessing the BLOB, but simplifies tuple processing
Long Records (3)

BLOBs can be stored in a B-Tree like fashion

- (relative) offset is search key
- allows for accessing and updating arbitrary parts
- very flexible and powerful
- but might be over-sophisticated
- SQL does not offer this interface anyway
Long Records (4)
Using an extent list is simpler

- real tuple points to BLOB tuple
- BLOB tuple contains a header and an extent list
- in worst case the extent list is chained, but should rarely happen
- extent list only allows for manipulating the BLOB in one piece
- but this is usually good enough
- hash and length to speed up comparisons
The Classical Architecture

Access

Long Records (5)

It makes sense to optimize for short BLOBs/CLOBs

- users misuse BLOBs/CLOBs
- they use CLOB to avoid specifying a maximum length
- but most CLOBs are short in reality
- on the other hand some BLOBs are really huge
- the DBMS cannot know
- so BLOBs can be arbitrary large, but short BLOBs should be more efficient

Approach:

1. BLOBs smaller than TID are encoded in BLOB TID
2. BLOBs smaller than page size are stored in BLOB record
3. only larger BLOBs use the full mechanism
Index Structures

Data is often indexed

- speeds up lookup
- de-facto mandatory for primary keys
- useful for selective queries

Two important access classes:

- point queries
  find all tuples with a given value (might be a compound)
- range queries
  find all tuples within a given value range

Support for more complex predicates is rare.
B-Tree

B-Trees (including variants) are the dominant data structure for external storage.

Classical definition:

- a B-Tree has a degree $k$
- each node except the root has at least $k$ entries
- each node has at most $2k$ entries
- all leaf nodes are at the same depth
B-Tree (2)

Example:

The ● is the TID of the corresponding tuple.
**B⁺-Tree**

Most DBMS use the B⁺-Tree variant:

- key+TID only in leaf nodes
- inner nodes contain separators, might or might not occur in the data
- increases the fanout of inner nodes
- simplifies the B-Tree logic

---

**B⁺-Tree**

with $k = 2$, $h = 3$

---

- $K_{49}$
- $K_{25}$ $K_{35}$
- $K_{02}$ $K_{03}$ $K_{16}$ $K_{25}$
- $K_{36}$ $K_{41}$ $K_{43}$ $K_{47}$
- $K_{26}$ $K_{29}$ $K_{35}$
- $K_{58}$ $K_{90}$
- $K_{91}$ $K_{95}$
- $K_{25}$ $K_{43}$ $K_{47}$
- $K_{86}$ $K_{88}$
Page Structure

Inner Node:
- LSN for recovery
- upper page of right-most child
- count number of entries
- key/child key/child-page pairs
... ...

Leaf Node:
- LSN for recovery
- ~0 leaf node marker
- next next leaf node
- count number of entries
- key/tid key/TID pairs
... ...

Similar to slotted pages for variable keys.
Operations - Lookup

Lookup a search key within the $B^+$-tree:

1. start with the root node
2. is the current node a leaf?
   - if yes, return the current page (locate the entry on it)
3. find the first entry $\geq$ search key (binary search)
4. if no such entry is found, go the upper, otherwise go to the corresponding page
5. continue with 2

Lookup can return a concrete entry or just the position on the appropriate leaf page (depends on usage pattern).
Operations - Insert

Insert a new entry into the $B^+$-tree:

1. *lookup* the appropriate leaf page
2. is there free space on the leaf?
   - if yes, insert entry and stop
3. split the leaf into two, insert entry on proper side
4. insert maximum of left page as separator into parent
5. if the parent overflow, split parent and continue with 4
6. create a new root if needed
Operations - Delete

Remove an entry from the $B^+$-tree:

1. lookup the appropriate leaf page
2. remove the entry from the current page
3. is the current page at least half full?
   ▶ if yes, stop
4. is the neighboring page more than half full?
   ▶ if yes, balance both pages, update separator, and stop
5. merge neighboring page into current page
6. remove the separator from the parent, continue with 3

Most systems simplify the delete logic and accept under-full pages.
Operations - Range Scan

Read all entries within a (start,stop) range

1. lookup the start value
2. enumerate subsequent entries on the current page
3. use the next pointer to find the next page
4. stop once the stop value is reached

Very efficient, in particular if leaf nodes are consecutive on disk.
Indexing Multiple Attribute

Compound keys are compared lexicographically:

$$(a_1, a_2) < (b_1, b_2) \iff (a_1 < b_1) \lor (a_1 = b_1 \land a_2 < b_2)$$

Otherwise compound keys are quite similar to atomic keys.

- when all attributes are bound, difference is minor
- if only a prefix is bound, the suffix is specified as range

$$a_1 = 5 \Rightarrow (5, -\infty) \leq (a_1, a_2) \leq (5, \infty)$$
Indexing Non-Unique Values

Users can create indexes on any attributes
- not necessarily unique
- in fact might contain millions of duplicates
- main problem: index maintenance
- how to locate a tuple for update/delete?

Solution: only index unique values
- append TID to non-key attributes
- TID works as a tie-breaker
- increases space consumption a bit
- but guarantees $O(\log n)$ access
Concurrent Access

How to handle concurrent access?

- simple page locking/latching is not enough
- will protect against “simple” (single page) changes
- but pages depend upon each other (pointers)

The classical technique is *lock coupling*

- a thread latches both the page and its parent page
- i.e., latch the root, latch the first level, release the root, latch the second level etc.
- prevents conflicts, as pages can only be split when the parent is latched
- no deadlocks, as the latches are ordered
Concurrent Access (2)

But what about inserts?

- when a leaf is split, the separator is propagated up
- might go up to the root
- but we have only locked one parent

Lock coupling up is not an option (deadlocks)

One way around it: “safe” inner pages

- while going down, check if the inner page has enough space for one more entry
- if not, split it
- ensures that we never go up more than one step
Concurrent Access (3)

Alternative: restart

1. first try to insert using simple lock coupling
2. if we do not have to split the inner node everything is fine
3. otherwise release all latches
4. restart the operation, but now keep all latches up to the root
5. all operations can be executed safely now

- greatly reduces concurrency
- but should happen rarely
- simpler to implement, in particular for variable-length keys
B-link Trees

- lock coupling latches two nodes at a time
- seems cheap, but effectively it locks hundreds (all children of the parent node)
- it would be nicer to lock only one page

For pure lookups that is possible when adding \textit{next} pointers to inner nodes:

1. latch a page, find the child page, release the page
2. latch the child page
3. might have been split in between, check neighboring pages

Requires some care when deleting.
Bulkloading

How to build an B-tree for a large amount of data?
- repeated inserts are inefficient
- a lot of random I/O
- pages are touched multiple times

Instead: sort the data before inserting
- now inserts become more efficient
- good locality

But we can do even better.
Bulkloading (2)

To construct an initial $B^+$-Tree:

1. sort the data
2. spool data into leaf pages
   ▶ fill the pages completely
   ▶ remember largest value (separator) in each page in a temp file
3. spool the separators into inner pages
   ▶ fill the pages completely
   ▶ remember largest value (separator) in each page in a temp file
4. repeat 3 until only one inner page remains (root)

Produces a compact, clustered $B^+$-tree.
Bulkloading (3)

Existing $B^+$-trees are a bit more problematic, but can still be updated in bulk

1. sort the data
2. merge the data into the existing tree
3. form pages, remember separators, etc.
4. start a new chunk once a page would contain only entries from the original tree
5. merge in the separators as above

Minimizes I/O, but destroys clustering. Usually a good compromise.
Partitioned B-Tree

Bulk operations are fine if they are rare, but they are disruptive

- usually the B-tree has to be take offline
- the new cannot be queries easily
- existing queries must be halted

Basic idea: *partition* the B-tree

- add an artificial column in front
- creates separate partitions with the B-tree
Partitioned B-Tree (2)

Benefits:
- partitions are largely independent of each other
- one can append to the “rightmost” partition without disrupting the rest
- the index stays always online
- partitions can be merged lazily
- merge only when beneficial

Drawbacks:
- no “global” order any more
- lookups have to access all partitions
- deletion is non-trivial (”anti-matter”)
Variable Length Records
So far B-trees are defined for fixed-length keys
- all nodes have between $k$ and $2k$ entries
- simplifies life considerably
- e.g., we “know” if an inner node is full

But in reality, entries can be variable length
- strings
- variable-length encoding, NULL
- compounds of these

Usually keys are *opaque*. The B-tree does not understand the structure. (One could special-case single strings).
Variable Length Records (2)

Variable length keys are problematic. Consider the following example:

```
a bbbbbbbb c dddddddd e ffffffff g hhhhhhhh
i jjjjjjjj k llllllll m nnnnnnnn o
```

All entries have either length 1 or length 8.
Variable Length Records (2)

Variable length keys are problematic. Consider the following example:

```
<table>
<thead>
<tr>
<th>a</th>
<th>bbbbbbbb</th>
<th>c</th>
<th>dddddd</th>
<th>e</th>
<th>fffffff</th>
<th>g</th>
<th>hhhhhhh</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>jjjjjjjjj</td>
<td>k</td>
<td>llllllll</td>
<td>m</td>
<td>nnnnnnnn</td>
<td>o</td>
<td></td>
</tr>
</tbody>
</table>
```

Variant A:
Variable Length Records (2)

Variable length keys are problematic. Consider the following example:

<table>
<thead>
<tr>
<th>a</th>
<th>bbbbbbbb</th>
<th>c</th>
<th>dddddddd</th>
<th>e</th>
<th>ffffffff</th>
<th>g</th>
<th>hhhhhhhh</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j jjjjjjjj</td>
<td>k</td>
<td>l lllllll</td>
<td>m</td>
<td>n nnnnnnn</td>
<td>o</td>
<td></td>
</tr>
</tbody>
</table>

Variant B:

Height 2 (ignores space consumption for pointers)
Variable Length Records (3)

Separator choice is crucial!
- affects fanout and space consumption
- all standard guarantees are off if normal algorithms are used for variable-length keys

Non-trivial issue
- greedy algorithms exist for the bulkloading case
- idea: recursively pick the smallest value as separator such that the resulting group sizes vary within a constant factor
- can also be extended to the dynamic case (rebuild as need), but non-trivial
- difficult, but gives good amortized bounds
Variable Length Records (4)

Minimal support: modify the split logic

1. when a page overflows, build the sorted list of all values
2. instead of picking the median value as separator, pick the smallest value within 20% around the median
3. for ties, prefer values closer to the median

- pragmatic solution, but not optimal
- can still degenerate
- avoids the worst mistakes
Prefix $B^+$-tree

A $B^+$-tree can contain separators that do not occur in the data

We can use this to save space:

- choose the smallest possible separator.
- no change to the lookup logic required
Prefix $B^+$-tree (2)

We can do even better by factoring out a common prefix:

- only one prefix per page
- the change to the lookup logic is minor
- the lookup key itself is adjusted
- sometimes only inner nodes, to keep scans cheap
Prefix B⁺-tree (3)

The lexicographic sort order makes prefix compression attractive:

- neighboring entries tend to differ only at the end
- a common prefix occurs very frequently
- not only for strings, also for compound keys etc.
- in particular important if partitioned B-trees
- with big-endian ordering any value might get compressed
Hash-Based Indexes

In main memory a hash table is usually faster than a search tree
- compute a hash-value $h$, compute a slot (e.g., $s = h \mod |T|$, access the table $T[s]$
- promises $O(1)$ access
- (if everything works out fine)

A DBMS could profit from this, too. But:
- random I/O is very expensive on disk
- collisions are problematic (e.g., when chaining)
- rehashing is prohibitive

But there are hashing schemes for external storage.
Hash-Based Indexes (2)

Hash indexes are not as versatile as tree indexes:

- only support point query
- range queries are very problematic
- order preserving hashing exists, but is questionable
- quality of the hash function is critical

As a consequence, mainly useful for primary key indexes

- unique keys
- key collisions would be very dangerous
- how to delete a tuple with an indexes attribute of there are 1 million other tuples with the same value?
- can be fixed by separate indexing within duplicate values (complicated)
Extendible Hashing
A central problem of hashing schemes on disk are the table size
- hard to known beforehand
- too small $\Rightarrow$ too many collisions
- chaining is expensive on disk
- too large $\Rightarrow$ waste of space
- would have to grow over time

Traditional solution for main memory: rehashing
- re-map items to hash table sizes
- involves touching every item
- poor locality, a lot of random I/O
- prohibitive for disk

Idea: Allow for growing the hash table without rehashing by sharing table entries.
Extendible Hashing (2)

- hash table size is always a power of 2
- hash table points to buckets (pages)
- multiple table entries can point to the same bucket
- but always systematically (buddy systems)
- thus, the “depth” of the buckets varies
Extendible Hashing (3)

- when a bucket overflows, it is split
- if not a maximum depth, the depth is increased
- achieved by de-sharing slot entries
- one more bit becomes relevant
- items are distributed according to hash values
Extendible Hashing (4)

- if the depth cannot be increased, the table is doubled
- other buckets are unaffected
- entries are duplicated, resulting in new sharing
- new buckets are linked as usual
Extendible Hashing (5)

- once a maximum table size is reached start chaining
- ideally occurs rarely, traversing chains is expensive
- optionally one can balance chaining vs. table growth via load factor
Extendible Hashing (6)

Advantages:
- ideally exactly two page accesses per lookup
- less than in a B-tree
- table can grow independent of existing buckets
- no need for re-hashing

Disadvantages:
- table growth is a very invasive operation
- large steps in space consumption
- what about hash collisions?
- same care is needed to avoid extreme table growth
Linear Hashing

- linear hashing avoids the exponential directory growth
- it starts with a regular hash table with buckets
Linear Hashing (2)

- when a bucket overflows it uses chaining
- degrades performance, but ok if the chain is short
Linear Hashing (3)

- triggered by load factor or chain length a bucket is split
- chains are re-integrated into the bucket
- only one (i.e., the next) bucket is split, the rest remains untouched
- the range of buckets $[1, k]$ has been split, the range $[k, n]$ is unsplit (the range $[n, n + 2k - 2]$ contains the second halves)
- the directory grows page by page
Linear Hashing (3)

- more buckets are split on demand
- at some point all buckets have been split once
- then the cycle starts anew
Linear Hashing (4)

Advantages:
- avoids the disruptive directory growth of EH
- index grows linearly
- amortized the index structure is nice

Disadvantages:
- chaining hurts performance
- it can take a while until chains are re-integrated
- page allocation for the directory problematic
Multi-Level Extendible Hashing

For uniform distributions the EH directory is nice:

![Diagram of uniform distribution]

But data skew causes poor space utilization

![Diagram of skewed distribution]

The directory size explodes.

- skew is an unfortunate reality
Multi-Level Extendible Hashing (2)

Basic idea: construct a tree of hash tables

- the next level uses the next $k$ bits
- node size is page size
- additional page faults, but fanout is very large
- only the heavily used parts get additional levels
Multi-Level Extendible Hashing (2)

Problem: space utilization

- now uniform is the worst case
- all buckets will overflow at the same time
- second-level hash tables will be nearly empty
- leads to poor space utilization (and poor performance)
Multi-Level Extendible Hashing (3)
Instead: share inner page between buddies
Multi-Level Extendible Hashing (4)

We can get additional bits by splitting the inner page
Multi-Level Extendible Hashing (5)

In fact here we can even move buddies up again.
Multi-Level Extendible Hashing (6)

- uses a buddy system (boundaries are powers of 2)
- bit width etc. derived implicitly
- pointer structure contains enough information
- results in large fanout

It has some nice properties
- naturally adapts to data skew
- directory growth is not a problem
- but additional page accesses
- tree is very shallow, however
Bitmap Indexes

Classical indexes do not handle unselective predicates very well

- large fraction of tuples is returned
- index access is more expensive than a table scan
- but a combination of predicates might be selective
- index intersection would help, but is still expensive
- predicates might also contain disjunctions etc.

Example: $\sigma_{(a=2 \lor b=5) \land (c \neq 1)}(R)$

Could be answered by B-trees, but often not very efficient.
Bitmap Indexes (2)

When the attribute domain is small, it can be indexed using **bitmap indexes**

<table>
<thead>
<tr>
<th>tid1</th>
<th>tid2</th>
<th>tid3</th>
<th>tid4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tid1</th>
<th>tid2</th>
<th>tid3</th>
<th>tid4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- one bitmap for every attribute value
- index intersections become bit operations
- very efficient
- remaining ones indicate matching tuples
Bitmap Indexes (3)

- bitmap indexes are compact (one bit per tuple)
- (plus tid directory if needed)
- and are usually sparse
- can be compressed very well
- run-length encoding in particular attractive
- intersection can be performed on the compressed form

Often outperforms other indexes for unselective attributes.
Small Materialized Aggregates

- Data is usually stored in physical chunks.
- Pages, or chunks of pages.
- Clustering usually insertion order.
- Older chunks tend to remain static.
- We can cheaply pre-computed (i.e., cache) some info.

Some useful aggregates:

<table>
<thead>
<tr>
<th>min</th>
<th>max</th>
<th>sum</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>1500</td>
<td>1000</td>
</tr>
</tbody>
</table>
Small Materialized Aggregates (2)

Before scanning a chunk, we examine the aggregate

- some predicates can be evaluated on the pure aggregate
- only for the current chunk, of course
- in particular skipping chunks is often possible
- aggregates can be directly re-used
- greatly saves I/O

What about updates?

- small materialized aggregates are like a cache
- can be updated eagerly or lazily
- an invalidation flag is enough
Multi-Dimensional Indexing

- huge field
- R-tree, grid file, pyramid schema, index intersection, ...
- hundreds of approaches
- we do not discuss them here

But: remember the **curse of dimensionality**

- we can only index a relative low number of dimensions
- for higher dimensions, range queries/proximity queries fail
- scan becomes faster than index structures
Transactions and Recovery
Transactions and Recovery

DBMSs offer two important concepts:

1. transaction support
   - a sequence of operations is combined into one compound operation
   - transactions can be execution concurrently with well defined semantics

2. recovery
   - the machine/DBMS/user code can crash at an arbitrary point in time, errors can occur, etc.
   - the recovery component ensures that no (committed) data is lost, instance is consistent

Implementation of both is intermingled, therefore we consider them together.
Why Transactions?

Transfer money from account A to account B

- read the account balance of A into the variable $a$: \texttt{read}(A,a);
- reduce the balance by EURO 50,–: $a := a - 50$;
- write back the new account balance: \texttt{write}(A,a);
- read the account balance of B into the variable $b$: \texttt{read}(B,b);
- increase the balance by EURO 50,–: $b := b + 50$;
- write back the new account balance: \texttt{write}(B,b);

Many issues here: crashes, correctness, concurrency, ...
Operations

- **begin of transaction (BOT):**
  - marks the begin of transaction
  - in SQL: begin transaction
  - often implicit

- **commit:**
  - terminates a successful transaction
  - in SQL: commit [transaction]
  - all changes are permanent now

- **abort:**
  - terminates an unsuccessful transaction
  - in SQL: rollback [transaction]
  - undoes all changes performed by the transaction
  - might be triggered externally

All transactions either commit or abort.
ACID

Transactions should offer ACID properties:

- **Atomicity**
  - the operations are either executed completely or not at all

- **Consistency**
  - a transaction brings a database instance from one consistent state into another one

- **Isolation**
  - currently running transactions are not aware of each other

- **Durability**
  - once a transaction commits successfully, its changes are never lost
Transactions and Recovery

The concept of *recovery* is related to the *transaction* concept:

- the DBMS must handle a crash at an arbitrary point in time
- first, the DBMS data structures must survive this
- second, transaction guarantees must still hold
- Atomicity
  - in-flight transactions must be rolled back at restart
- Consistency
  - consistency guarantees must still hold
- Durability
  - committed transactions must not be lost, even though data might still be in transient memory

Sometimes the dependency is mutual

- Isolation
  - some DBMS use the recovery component for transaction isolation
Technical Aspects

The logical concept *transactions* and *recovery* can be seen under (largely orthogonal) technical aspects:

- concurrency control
- logging

As we will see, both are relevant for both logical concepts.
Multi User Synchronization

- executing transactions (TA) serialized is safe, but slow
- transactions are frequently delayed (wait for disk, user input, ...)
- in serial execution, would block all other TAs
- concurrent execution is desirable for performance reasons

But: simple concurrent execution causes a number of problems.
Lost Update

The result of transaction $T_1$ is lost.
Dirty Read

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{bot} & \quad \text{bot} \\
\leftarrow & \quad r_2(x) \\
& \quad w_2(x) \\
& \quad r_1(x) \\
w_1(y) & \quad \leftarrow \\
\text{commit} & \quad \leftarrow \\
\leftarrow & \quad \text{abort}
\end{align*}
\]

\(T_1\) reads an invalid value \(x\).
Non-Repeatable Read

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{bot} & \text{bot} \\
\text{\(r_1(x)\)} & \text{\(w_2(x)\)} \\
\text{\(\leftarrow\)} & \text{commit} \\
\text{\(r_1(x)\)} & \leftarrow \\
\ldots & \\
\end{array}
\]

\(T_1\) reads the value \(x\) twice, with different results.
**Phantom Problem**

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{bot} & \text{bot} \\
\text{select count(*)} & \text{select count(*)} \\
\text{from R;} & \text{from R;} \\
\leftarrow & \leftarrow \\
\end{array}
\]

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{bot} & \text{insert into R . . . ;} \\
\text{commit} & \\
\end{array}
\]

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{select count(*)} & \text{select count(*)} \\
\text{from R;} & \text{from R;} \\
\ldots & \\
\end{array}
\]

\(T_1\) sees a new tuple during hit second access.
Serial Execution

These problems vanish with *serial* execution
- a transaction always controls the whole DBMS
- no conflicts possible
- but poor performance

Instead: execute transaction as if they were serial
- if they behave as if they were serial they cause no problems
- concept is called *serializable*
- requires some careful bookkeeping
Formal Definition of a Transaction

- Possible operations of a TA $T_i$
  - $r_i(A)$: read the data item $A$
  - $w_i(A)$: write the data item $A$
  - $a_i$: abort
  - $c_i$: commit successfully

- $bot$: begin of transaction (implicit)
Formal Definition of a Transaction (2)

- A TA $T_i$ is a partial order of operations with the order relation $<_i$ such that
  - $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$
  - $a_i \in T_i$, iff $c_i \notin T_i$
  - Let $t$ be $a_i$ or $c_i$. Then for all other operations $p_i$: $p_i <_i t_i$
  - If $r_i[x] \in T_i$ and $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$
Example

- transactions are often drawn as directed acyclic graphs (DAGs)

\[
\begin{align*}
&\text{r}_2[x] \\
&\text{w}_2[z] \quad \text{c}_2 \\
&\text{r}_2[y]
\end{align*}
\]

- \( r_2[x] <_2 w_2[z], w_2[z] <_2 c_2, r_2[x] <_2 c_2, r_2[y] <_2 w_2[z], r_2[y] <_2 c_2 \)

- transitive relationships are contained implicitly
Schedules

- multiple transactions can be executed concurrently
- this is captured by a *schedule*
- a schedule orders the operations of the TAs relative to each other
- due to the concurrent execution of operations the schedule defines only partial ordering
Conflicting Operations

- operations that are conflicting must not be executed in parallel
- two operations are in conflict if both operate on the same data item and at least one of the two is a write operation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_j$</td>
<td>$r_i[x]$</td>
<td>$w_i[x]$</td>
</tr>
<tr>
<td>$r_j[x]$</td>
<td></td>
<td>¬</td>
</tr>
<tr>
<td>$w_j[x]$</td>
<td>¬</td>
<td>¬</td>
</tr>
</tbody>
</table>
Definition of a Schedule

- Let $T = \{ T_1, T_2, \ldots, T_n \}$ be a set of transactions.
- A schedule $H$ over $T$ is a partial order with order relation $<_H$, such that:
  - $H = \bigcup_{i=1}^{n} T_i$
  - $<_H \supseteq \bigcup_{i=1}^{n} <_i$
  - For all conflicting operations $p, q \in H$ the following holds: either $p <_H q$ or $q <_H p$
Example

\[
H = \begin{align*}
& r_2[x] \rightarrow w_2[y] \rightarrow w_2[z] \rightarrow c_2 \\
& r_3[y] \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow w_3[z] \rightarrow c_3 \\
& r_1[x] \rightarrow w_1[x] \rightarrow c_1
\end{align*}
\]
(Conflict-)Equivalence

- The schedules $H$ and $H'$ are \textit{(conflict-)equivalent} ($H \equiv H'$), if:
  - both contain the same set of TAs (including the corresponding operations)
  - both order conflicting operations of non-aborted TAs in the same way
- the general idea is that executing conflicting operations in the same order will produce the same result
Example

\[ r_1[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \]

\[ \equiv r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \]

\[ \equiv r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \]

\[ \not\equiv r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1 \]
Serializability

- serial schedules are safe, therefore we are interested in schedules with similar properties
- in particular we want schedules that are equivalent to a serial schedule
- such schedules are called *serializable*
Serializability (2)

- Definition
  - The committed projections $C(H)$ of a schedule $H$ contains only the committed TAs
  - A schedule $H$ is serializable, if $\exists H_s$ such that $H_s$ is serial and $C(H) \equiv H_s$. 
Serializability (3)

- How to check for serializability?
- A schedule $H$ is serializable if and only if the serializability graph $SG(H)$ is acyclic.
Serializability Graph

- The serializability graph $SG(H)$ of a schedule $H = \{ T_1, \ldots, T_n \}$ is a directed graph with the following properties:
  - the nodes are formed by the committed transactions from $H$
  - two TAs $T_i$ and $T_j$ are connected by an edge from $T_i$ to $T_j$ if there exist two operations $p_i \in T_i$, $q_j \in T_h$ such that $p_i$ and $q_j$ are in conflict and $p_i <_H q_j$. 


Example

- Schedule $H$

$$H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3$$

- $SG(H)$

$$SG(H) = \begin{align*}
T_2 \\
T_1 \\
T_3
\end{align*}$$
Example (2)

- $H$ is serializable
- equivalent serial schedules

\[
H_s^1 = T_1 \mid T_2 \mid T_3 \\
H_s^2 = T_1 \mid T_3 \mid T_2 \\
H \equiv H_s^1 \equiv H_s^2
\]
Example (3)

\[
\begin{align*}
    r_1[x] & \rightarrow w_1[x] \rightarrow w_1[y] \rightarrow c_1 \\
    H = \quad \downarrow & r_2[x] \rightarrow w_2[y] \rightarrow c_2 \\
    r_3[x] & \rightarrow w_3[x] \rightarrow c_3
\end{align*}
\]

\[
T_3 \\
SG(H) = T_2 \\
T_1
\]
Example (4)

- $H$ is serializable
- equivalent serial schedules

\[ H_s^1 = T_2 \mid T_1 \mid T_3 \]

\[ H \equiv H_s^1 \]
Example (5)

\[ H = \begin{align*}
   w_1[x] & \rightarrow w_1[y] & \rightarrow c_1 \\
   \uparrow & & \downarrow \\
   r_2[x] & \rightarrow w_2[y] & \rightarrow c_2
\end{align*} \]

\[ \text{SG}(H) = T_1 \iff T_2 \]

- \( H \) is not serializable
Additional Properties of a Schedule

- Besides serializability, other properties are desirable, too:
  ▶ recoverability
  ▶ avoiding cascading aborts: ACA
  ▶ strictness

Recoverability is required for correctness, the others are more nice to have (but are crucial for some implementations).
Additional Properties of a Schedule (2)

- Before looking at more properties, we define the reads-from relationship

- A TA $T_i$ read (data item $x$) from TA $T_j$, if
  - $w_j[x] < r_i[x]$
  - $a_j \not< r_i[x]$
  - if $\exists w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$

- a TA can read from itself
Recoverability

- A schedule is *recoverable*, if
  - Whenever TA $T_i$ reads from another TA $T_j$ ($i \neq j$) and $c_i \in H$, then $c_j < c_i$
- the TAs must adhere to a certain commit order
- non-recoverable schedules may cause problems with C and/or D of the ACID properties
Recoverability (2)

\[ H = w_1[x] \ r_2[x] \ w_2[y] \ c_2 \ a_1 \]

- \( H \) is not recoverable
- this has some unfortunate consequences:
  - if we keep the updates from \( T_2 \) then the data is inconsistent (\( T_2 \) has read data from an aborted transaction)
  - if we undo \( T_2 \), the we change committed data
## Cascading Aborts

<table>
<thead>
<tr>
<th>step</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>$w_1[x]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td>$r_2[x]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>$w_2[y]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td></td>
<td>$r_3[y]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
<td>$w_3[z]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td></td>
<td></td>
<td>$r_4[z]$</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td></td>
<td></td>
<td>$w_4[v]$</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r_5[v]$</td>
</tr>
<tr>
<td>9.</td>
<td>$a_1$ (abort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cascading Aborts (2)

- A schedule *avoids cascading aborts*, if the following holds
  - whenever a TA $T_i$ reads from another TA $T_j$ ($i \neq j$), then $c_j < r_i[x]$
- We must only read from transactions that have committed already.
Strictness

- A schedule is *strict*, if the following holds
  - for any two operations $w_j[x] < o_i[x]$ (with $o_i[x] = r_i[x]$ or $w_i[x]$) either $a_j < o_i[x]$ or $c_j < o_i[x]$
- We must only read from committed transactions, and only overwrite changes made by committed transactions.
Strictness (2)

- Only strict schedules allow for physical logging during recovery

\[ x = 0 \]
\[ w_1[x, 1] \text{ before image of } T_1: 0 \]
\[ x = 1 \]
\[ w_2[x, 2] \text{ before image of } T_2: 1 \]
\[ x = 2 \]
\[ a_1 \]
\[ c_2 \]

When aborting \( T_1 \) \( x \) would incorrectly be set to 0.
Classification of Schedules

SR: serializable, RC: recoverable, ACA: avoids cascading aborts, ST: strict
Scheduler

- the scheduler orders incoming operations such that the resulting schedule is serializable and recoverable.
- options:
  - execute (immediately)
  - reject
  - delay
- two main classes of strategies:
  - pessimistic
  - optimistic
Pessimistic Scheduler

- scheduler delays incoming operations
- for concurrent operations, the scheduler picks a safe execution order
- most prominent example: lock-based scheduler (very common)
Optimistic Scheduler

- scheduler executes incoming operations as quickly as possible
- might have to rollback later
- most prominent example: time-stamp based scheduler
Lock-based Scheduling

- The main idea is simple:
  - each data item has an associated lock
  - before a TA \( T_i \) accesses a data item, it must acquire the associated lock
  - if another TA \( T_j \) holds the lock, \( T_i \) has to wait until \( T_j \) releases the lock
  - only one TA may hold a lock (and access the corresponding data item)

- how to guarantee serializability?
Two-Phase Locking

- Abbreviated as 2PL
- Two lock modes:
  - S (shared, read lock)
  - X (exclusive, write lock)

Compatibility matrix:

<table>
<thead>
<tr>
<th>acquired lock</th>
<th>none</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>√</td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td>X</td>
<td>√</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Definition

- before accessing a data item a TA must acquire the corresponding lock
- a TA must not request a lock that it already holds
- if a lock cannot be granted immediately, the TA is put into a wait queue
- a TA must not acquire new locks once it has released a lock (two phases)
- at commit (or abort) all held locks must be released
Two Phases

- growing phase: locks are acquired, but not released
- shrinking phase: locks are released, but not acquired
## Concurrency with 2PL

<table>
<thead>
<tr>
<th>Schritt</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BOT</td>
<td>BOT</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>lock( X[x] )</td>
<td>lock( S[x] )</td>
<td>( T_2 ) has to wait</td>
</tr>
<tr>
<td>3.</td>
<td>( r[x] )</td>
<td>( r[x] )</td>
<td>( T_2 ) wakes up</td>
</tr>
<tr>
<td>4.</td>
<td>( w[x] )</td>
<td>( r[y] )</td>
<td>( T_2 ) has to wait</td>
</tr>
<tr>
<td>5.</td>
<td>unlock( X[x] )</td>
<td>( lock( S[y] ) \</td>
<td>( T_2 ) wakes up</td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td>( w[y] )</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td>unlock( X[y] )</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td></td>
<td>unlock( S[x] )</td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td></td>
<td>unlock( S[y] )</td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td></td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
Strict 2PL

- 2PL does not avoid cascading aborts
- extension to strict 2PL:
  - all locks must be held until the end of transaction
  - avoids cascading aborts (the schedule is even strict)
Strict 2PL (2)
Lock Manager

locks are typically organized in a hash table
Lock Manager (2)

Traditional architecture:

- one mutex per lock chain
- within the lock, separate locking/waiting mechanism
- syncing chain mutex/lock latch needs some care to maximize concurrency
- lock includes ownership and lock mode information

Separate per-transaction chaining

- needed for EOT
- no latching required
- but: can only be embedded easily for exclusive locks
- in general: keep the list external
Lock Manager (3)

One problem: EOT

- all locks have to be released
- lock list is available
- but puts a lot of stress on the lock manager
- chains may be scanned and locked repeatedly
- one option: lazy removal of lock entries
- allows for EOT without locking the chains
Reducing the Lock Size

Locks a relatively expensive
- typically 64-256 bytes per lock
- thousands, potentially millions of locks
- space utilization becomes a problem
- commercial DBMS limit the amounts of locks

One solution: use less locks
- space/granularity trade-off
- leads to MGL (as we will see)
- may cause unnecessary aborts

Other option: reduce the size of locks
Reducing the Lock Size (2)

- standard locks contain a wait mechanism
- but when we use strict 2PL, we wait for transactions anyway
- it is sufficient to contain the owner in the lock
- we always wait for the owner
- shared locks are a bit problematic (requires some effort)

| 64 bit key | 32 bit owner | 32 bit status |

- status include lock mode, pending writes, etc.
- concurrently held require some care (linked list, spurious wakeups, etc.)
- but that is fine if the lists are short
## Deadlocks

- **Example:**

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bot</td>
<td></td>
<td>bot</td>
</tr>
<tr>
<td>lockX$_1$(a)</td>
<td></td>
<td>lockS$_2$(b)</td>
</tr>
<tr>
<td>$w_1$(a)</td>
<td></td>
<td>$r_2$(b)</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td></td>
</tr>
<tr>
<td>lockX$_1$(b)</td>
<td>$\leftrightarrow$</td>
<td>lockS$_2$(a)</td>
</tr>
</tbody>
</table>
Deadlock Detection

- no TA should have to wait “forever”
- one strategy to avoid deadlocks are time-outs
  - finding the right time-out is difficult
- a precise method analyzes the waits-for graph
  - TAs form node, edges are induced by waits-for relations
  - if the graph is cyclic we have a deadlock
Waits-for graph

- Example

- the waits-for graph is cyclic, i.e., we have a deadlock
- we can break the cycle by aborting $T_2$ or $T_3$
Implementing Deadlock Detection

- timeouts are simple, fast, and crude
- cycle detection is precise but expensive

One alternative: use a hybrid approach

- use a short timeout
- after the timeout triggered, start the graph analysis
- build the wait-for graph on demand

Keeps the common case fast, deadlock detection is only slightly delayed.
Online Cycle Detection

How to find cycles in a directed graph?

- simple solution: depth-first-search and mark
- we have a cycle if we meet a marked node
- problem: $O(n + m)$
- executed at every check

Better: use an online algorithm

- remembers information from last checks
- only re-computes if needed

Observation: a graph is acyclic if and only if there exists a topological ordering.
Online Cycle Detection (2)

- we start with an arbitrary topological ordering $<_T$
- when trying to add a restriction $B < A$, we perform a check

```ml
if $B <_T A$
  return true
marker[B]=2
if ¬ dfs(A,B)
  for each $V \in [A,B]$
    marker[V]=0
  return false
shift(A,B)
```

- dynamically updates the ordering
Online Cycle Detection (3)


\[
\text{dsf}(N, L) \\
\text{marker}[N] = 1 \\
\text{for each } V \text{ outgoing from } N \\
\text{if } V \leq_T L \\
\quad \text{if } \text{marker}[V] = 2 \\
\quad \quad \text{return false} \\
\quad \text{if } \text{marker}[V] = 0 \\
\quad \quad \text{if } \neg \text{dsf}(V, L) \\
\quad \quad \quad \text{return false} \\
\text{return true}
\]
Online Cycle Detection (4)

Update the ordering

\begin{align*}
\text{shift}(B,A) & \\
\text{marker}[B] & = 0 \\
shift & = 0 \\
L & = <> \\
\text{for each } V & \in [A,B] \\
\text{if } \text{marker}[V] & > 0 \\
L & = L \circ < V > \\
\text{shift} & = \text{shift} + 1 \\
\text{marker}[V] & = 0 \\
\text{else} & \\
\text{move } V & \text{ shift steps to the left} \\
\text{place the entries in } L & \text{ at } B - \text{shift}
\end{align*}
Multi-Granularity Locking

- (strict) 2PL solves the mentioned isolation problems, except the phantom problem
- the phantom-problem cannot be solved by standard locks, as we cannot lock something that does not exist
- we can solve this by using *hierarchical locks* (multi-granularity locking: MGL)
MGL

Database

Segments

Pages

Records
Additional Lock Modes for MGL

- $S$ (shared): read only
- $X$ (exclusive): read/write
- $IS$ (intention share): intended reads further down
- $IX$ (intention exclusive): intended writes further down the hierarchy
## Compatibility Matrix

<table>
<thead>
<tr>
<th>requested</th>
<th>current lock</th>
<th>none</th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>−</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>√</td>
<td>−</td>
<td>√</td>
<td>−</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>IS</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>−</td>
<td>√</td>
</tr>
<tr>
<td>IX</td>
<td></td>
<td></td>
<td>√</td>
<td>−</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>
Protocol

- Locks are acquired top-down
  - for a $S$ or $IS$ lock all ancestors must be locked in $IS$ or $IX$ mode
  - for a $X$ or $IX$ lock all ancestors must be locked in $IX$ mode
- locks are released bottom-up (i.e., only if no locks on descendants remain)
Example
Example (2)

Database

Segments

Pages

Records

(T₂, IS)

(T₁, IX)

(T₂, IS)

(T₄, IX)

(T₃, IX)

(T₅, S)

(T₁, X)

(T₂, S)

(T₃, X)

(T₄, IX)

(T₅, S)

(T₆, IS)

(T₅, IS)

(T₄, IX)

(T₅, IS)

(T₄, IX)

(T₅, IS)

(T₄, IX)
Example (3)

- TAs $T_4$ and $T_5$ are blocked
- we have no deadlock here, but deadlocks are possible with MGL, too.
Using MGL for Lock Management

Another important use for MGL: lock management

- most DBMSs cannot cope with a huge number of locks
- usually an upper bound on the number of locks
- but MGL can reduce the load
- we can reduce the locks by locking higher hierarchy levels
- and then release the descendant locks
- allows for scaling the number of locks

But: can easily lead to deadlocks/aborted transactions.
Preventing Phantom Problems without MGL

Another way to prevent the phantom problem: add a lock for the “next” tuple

- adds a lock for the “next” pseudo-tuple
- non-PK scans lock this tuple shared
- insert operations lock it exclusive
- prevents phantoms

But: we may want concurrent inserts

- another lock mode just for inserts
- if the TA scans+inserts, we really want exclusive
- gets a bit tricky
- but can be solved
Timestamp Based Approaches

- timestamp based synchronization is an alternative to locking
- each TA is assigned a unique timestamp
- each operation of the TA is uses this timestamp

Assignment of timestamps varies (eager, lazy, ...), the simplest case is order by BOT.
The scheduler uses the timestamps to order conflicting operations

- assume that $p_i[x]$ and $q_j[x]$ are conflicting operations
- $p_i[x]$ is executed before $q_j[x]$, iff the timestamp of $T_i$ is older than the timestamp of $T_j$
Timestamps (2)

- the scheduler annotates each data item \( x \) with the timestamp of the last operations on \( x \)
- timestamps are stored separately for each type of operation \( q \): \( \text{max-q-scheduled}(x) \)
- when the scheduler tries to execute an operator \( p \), the timestamp of \( p \) is compared to all \( \text{max-q-scheduled}(x) \) that conflict with \( p \)
- if the timestamp of \( p \) is older than any \( \text{max-q-scheduled}(x) \) the operations is rejected (and the TA aborted)
- otherwise \( p \) is executed and \( \text{max-p-scheduled}(x) \) is updated
Commit Order

- using the basic timestamp approach might produce non-recoverable schedules
- we can guarantee recoverability by committing TAs in timestamp order
- the commit of a TA $T_i$ is delayed as long as transaction from which $T_i$ has read are still active.

Ideally, timestamps are given out in commit order
- hard to know beforehand
- one alternative: transaction reordering
Limitations

Timestamps are used only relatively rarely
- does not avoid the phantom problem
- aborts TAs if there is any indication of problems
- every read operations is implicitly a write (updating the timestamps)

But it also has some strength
- can synchronize an arbitrary number of items (unlike locks)
- easy to distribute/parallelize

Might become more attractive considering current hardware trends.
Snapshot Isolation

- the DBMS has to keep track of all updates performed by a TA
- needed to undo a TA
- this information is usually available even after a TA committed
- therefore the DBMS can (conceptually) remove the effect of any TA

This can be used to isolate transaction:

- at BOT, the TA is assigned a timepoint $T$
- all committed changes before are visible
- all changes after $T$ are removed from the data view
- conceptually produces a snapshot of the data
Snapshot Isolation (2)

How to implement SI?
- makes use of the transaction log
- every page contains the LSN
- indicates the last change
- pages with old LSN can be read safely
- for pages with newer LSN the log is checked to eliminate recent changes
Snapshot Isolation (3)

Snapshot isolation has some very nice properties:

- no need for read locks (which could be millions)
- read operations never wait
- serializability (but see below)

Limitations:

- only safe for read-only transactions!
- a read-write transaction must not use snapshot isolation if the schedule has to be serializable
- still, many systems use snapshot isolation even for r/w TAs
Recovery

- a DBMS must not lose any data in case of a system crash
- main mechanisms of recovery:
  - database snapshots (backups)
  - log files
Recovery (2)

- A *database snapshot* is a copy of the database at a certain point in time.
- The *log file* is a protocol of all changes performed in the database instance.
- Obviously, the main data, the database snapshots, and the log-files should not be kept on the same machine...
System Failure

The Classical Architecture

Transactions and Recovery

{
State 1

backup

{\{T_i, T_j, T_k\}

State 2

(lost)

Restoration

restore

State 1

state 2

Log File

Snapshot

rollforward

State 1

State 2
Main Memory Loss

- problem: some TAs in \{T_j\} where still active, some committed already
- restart reconstructs state 2 + all changes by committed TAs in \{T_j\}
Abort a Transaction

- log files can also be used to undo the changes performed by an aborted TA
- the functionality is needed anyway (system crash)
- can be used for “normal” aborts, too

We now look more closely at the implementation.
Classification of Failures

- local failure within a non-committed transaction
  - effect of TA must undone
  - $R_1$ recovery

- failure with loss of main memory
  - all committed TAs must be preserved ($R_2$ recovery)
  - all non-committed TAs must be rolled back ($R_3$ recovery)

- failure with loss of external memory
  - $R_4$ recovery
Storage Hierarchy

DBMS Buffer

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A'</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>C'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

External Memory

write

read

PA

PC

PB

A'

D

C

B
Storage Hierarchy (2)

- Replacement strategies for buffer pages
  - \(\neg\text{steal}\): pages that have been modified by active transactions must not be replaced
  - \text{steal}: any non-fixed pages can be replaced if new pages have to be read in
Storage Hierarchy (3)

- write strategies for committed TAs
  - *force* strategy: changes are written to disk when a TA commits
  - ¬*force* strategy: changed pages may remain in the buffer and are written back at some later point in time
## Effects on Recovery

<table>
<thead>
<tr>
<th></th>
<th>force</th>
<th>¬force</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬steal</td>
<td>• no redo</td>
<td>• redo</td>
</tr>
<tr>
<td></td>
<td>• no undo</td>
<td>• no undo</td>
</tr>
<tr>
<td>steal</td>
<td>• no redo</td>
<td>• redo</td>
</tr>
<tr>
<td></td>
<td>• undo</td>
<td>• undo</td>
</tr>
</tbody>
</table>
Update Strategies

- **Update in Place**
  - each page corresponds to one fixed position on disk
  - the old state is overwritten

- **twin-block approach**

\[ P_0^A \mid P_1^A \mid P_0^B \mid P_1^B \mid P_0^C \mid P_1^C \mid \ldots \]

- **shadow pages**
  - only changed pages are replicated
  - less redundancy than with the twin-block approach
System Configuration

In the following we assume a system with the following configuration

- steal
- force
- update-in-place
- fine-grained locking
The ARIES protocol is a very popular recovery protocol for DBMSs

- The log file contains:
  - Redo Information: contains all information necessary to re-apply changes
  - Undo Information: contains all information necessary to undo changes
The log information stored written two times

- log file for fast access: R1, R2, and R3 recovery
- log archive: R4 recovery
Writing the Log (2)

- organization of the log ring-buffer:
Writing the Log (3)

- **Write Ahead Log Principle**
  - before a transaction is **committed**, all corresponding log entries must have been written to disk
  - before a modified page is written back to disk, all log entries involving this page must have been written to disk

- this is called *forcing* the log

Required for Durability.
Writing the Log (4)

Some care is needed when writing the log to disk

- disks are not byte addressable
- larger chunks, usually 512 bytes
- remember, the system may crash at any time
- partial writes to the last block are dangerous
- might require additional padding when forcing the log
- related problem: partial page writes

Some of these issues can be solved by hardware.
Restart after Failure

- TAs like $T_1$ are *winner* transactions: they must be replayed completely
- TAs like $T_2$ are *loser* transactions: they must be undone
Restart Phases

- **Analysis:**
  - determine the *winner* set of transactions of type $T_1$
  - determine the *loser* set of transactions of type $T_2$.

- **Repeating History:**
  - all operations contained in the log are applied to the database instance in the original order

- **Undo of Loser Transactions:**
  - the operations of *loser* transactions are undone in the database instance in reverse order
Restart Phases (2)

1. Analysis

2. Redo of all changes (*Winner* and *Loser*)

3. Undo of all changes from *Loser* transactions
Structure of Log Entries

[LSN, TA, PageID, Redo, Undo, PrevLSN]

- Redo:
  - physical logging: after image
  - logical logging: code that constructs the after image from the before image

- Undo:
  - physical logging: before image
  - logical logging: code that constructs the before image from the after image
Structure of Log Entries (2)

- **LSN (Log Sequence Number),**
  - a unique number identifying a log entry
  - *LSNs* must grow monotonically
  - allows for determining the chronological order of log entries
  - typical choice: offset within log file (i.e., implicit)

- **TA**
  - transaction ID of the transaction that performed the change
Structure of Log Entries (3)

- **PageID**
  - the ID of the page where the update was performed
  - if a change affects multiple pages, multiple log records must be generated

- **PrevLSN,**
  - pointer to the previous log entry of the corresponding transactions
  - needed for performance reasons

Note: often there is a certain asymmetry: physical redo (one page), logical undo (multiple pages)
### Example

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>BOT</strong></td>
<td><strong>BOT</strong></td>
<td>[LSN, TA, PageID, Redo, Undo, PrevLSN]</td>
</tr>
<tr>
<td>2</td>
<td>$r(A, a_1)$</td>
<td>$r(C, c_2)$</td>
<td>[#1, $T_1$, BOT, 0]</td>
</tr>
<tr>
<td>3</td>
<td>$a_1 := a_1 - 50$</td>
<td>$c_2 := c_2 + 100$</td>
<td>[#2, $T_2$, BOT, 0]</td>
</tr>
<tr>
<td>4</td>
<td>$w(A, a_1)$</td>
<td>$w(C, c_2)$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$r(B, b_1)$</td>
<td>$r(B, b_1)$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$b_1 := b_1 + 50$</td>
<td>$b_1 := b_1 + 50$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$w(B, b_1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>commit</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Phases - Analysis

- the log contains BOT, commit, and abort entries
- the log is scanned sequentially to identify all TAs
- when a *commit* is seen, the TA is a *winner*
- when a *abort* is seen, the TA is a *loser*
- TAs that neither commit nor abort are implicitly *loser*

Winner have to be preserved, loser have to be undone
The Phases - Redo

Redo brings the DB into a consistent state

- some changes might still be in main memory at the crash (/force)
- changes can be incomplete (e.g., B-tree split)
- but the log contains everything

Redo is done by one forward pass

- all log entries contain the affected page
- the pages contain LSN entries
- if the LSN of the page is less than the LSN of the entry, the operation must be applied
- the LSN is updated afterwards!
- allows for identifying the current state

Afterwards the DB has a known state.
The Phases - Undo

Eliminates all changes by loser transactions.
- during analysis, DBMS remembers last LSN of each transaction
- transactions that aborted on their own can be ignored (no “last operation”, all undone)
- active TAs have to be rolled back

Log is read backwards
- lastLSN pointers are used for skipping
- all encountered operations are undone
- produces new log entries (redo the undo)
Idempotent Restart

\[
\text{undo}(\text{undo}(\cdots(\text{undo}(a))\cdots)) = \text{undo}(a) \\
\text{redo}(\text{redo}(\cdots(\text{redo}(a))\cdots)) = \text{redo}(a)
\]
Idempotent Restart (2)

- CLRs (compensating log records) for undone changes
- #7’ is a CLR for #7
- #4’ is a CLR for #4
Log Entries after Restart

[#1, \(T_1, \text{BOT}, 0\)]
[#2, \(T_2, \text{BOT}, 0\)]
[#3, \(T_1, P_A, A--=50, A+=50, \#1\)]
[#4, \(T_2, P_C, C+=100, C--=100, \#2\)]
[#5, \(T_1, P_B, B+=50, B--=50, \#3\)]
[#6, \(T_1, \text{commit}, \#5\)]
[#7, \(T_2, P_A, A--=100, A+=100, \#4\)]

\langle \#7', T_2, P_A, A+=100, \#7, \#4 \rangle
\langle \#4', T_2, P_C, C--=100, \#7', \#2 \rangle
\langle \#2', T_2, \_ , \_ , \#4', 0 \rangle

- CLRs are marked by \langle \ldots \rangle
CLR

- A CLR is structured as follows:
  - LSN
  - TA
  - PageID
  - Redo information
  - PrevLSN
  - UndoNxtLSN (pointer to the next operation to undo)

- No undo information (redo only)

- PrevLSN/undoNxtLSN could be combined into one (prevLSN is not really needed)
Partial Rollback

- Steps 3 and 4 are rolled back
- necessary to implement save points within a TA
Checkpoints

- \( T_1 \) used to speed up restart
Checkpoints (2)

- transaction consistent:

- undo
- redo
- analysis
- checkpoint
- log

Diagram:

- Log
- Checkpoint
- Analysis
- Redo
- Undo
Checkpoints (3)

- action consistent:

```
Log: MinLSN

checkpoint

Analysis
Redo
Undo

MinLSN
```
Checkpoints (4)

T₁ | T₂ | T₃ | T₄

T₁, T₂, T₃, and T₄ represent checkpoints in the system. The diagram shows the timeline of transactions and checkpoints, with T₃ occurring after T₂ and before T₄. The crash line indicates a failure point after T₄.
Checkpoints (5)

- fuzzy checkpoints:

![Diagram of checkpoints]

- checkpoint
- Log
- MinDirtyPageLSN
- MinLSN
- Analysis
- Redo
- Undo
Fuzzy Checkpoints

- modified pages are not forced to disk
- only the page ids are recorded
- *Dirty Pages* = set of all modified pages
- *MinDirtyPageLSN*: the minimum LSN whose changes have not been written to disk yet
Set-Oriented Query Processing
Efficient Query Processing

Set-Oriented Query Processing

Motivation

During query processing, the DBMS tries to process whole *sets of data items* at a time

- “manual” programming is usually record oriented
- e.g., compare two records
- easy to understand, but this does not scale

Consider: intersecting two lists

- breaking it down into record-level operators is inefficient
- compares each record with each other record
- $O(n^2)$
- considering the complete lists in one step is more efficient
- $O(n \log n)$
Motivation (2)

Set-oriented processing has several advantages

- data can be pre-processed before processing
- sorting/hashing/index structures etc.
- amortizes over the set
- leads to more efficient algorithms
- easier to cope with memory limitations etc.
- easier parallelism
- ...

Algorithms tend to become more scalable, but also more involved.
The Algebraic Model

Query processing is usually expressed by relational algebra

- operators consumes zero or more relations, and produce one output relation
- inherently set (or rather: bag) oriented

\[
\begin{align*}
\text{students} & \quad \Join \quad \text{attend} \\
\text{lectures} & \quad \sigma_{\text{title} = \text{“DB”}}
\end{align*}
\]
Implementing the Algebraic Model

Operators are specified in a query agnostic manner:

- intersect
  - left
  - right
  - compare

Operator does not understand the query semantic. It only knows:

- *left* will produce a result set
- *right* will produce a result set
- *compare* compares two elements

Note: a scalable implementation will need more (e.g., $hashLeft, hashRight$), we ignore this for now.
Implementing the Algebraic Model (2)

The algebraic operators define the **abstract logic** of query processing primitives. The query specific parts are hidden in **subscripts**.

In particular:

- operators do not “know” the data types or byte size of input tuples
- they do not “understand” the content of a tuple
- they only specify the data flow and the control flow
- all query dependent operations are delegated to helper subscripts
- keeps the operator itself very generic

Note: sometimes operators are hinted with query specific info (e.g., a fixed tuple size) for performance reasons, but this is only a minor variation.
Implementing the Algebraic Model (3)
Example: intersectSorted(left, right, compare)

\( t_1 = \) next tuple from \( left \)
\( n = right \)

while input is not exhausted
  if \( n = left \)
    \( t_1 = \) next tuple from \( left \) else
    \( t_2 = \) next tuple from \( right \)
  \( c = compare(t_1, t_2) \)
  if \( c = 0 \)
    store \( t_1 \) as result
  else if \( c < 0 \)
    \( n = left \)
  else
    \( n = right \)

The code is independent from the concrete query.
Operator Composition

- each operator produces a set (bag/stream) of result tuples
- operators consume zero or more input sets
- usually assume nothing about their input
- therefore can be combined in an arbitrary manner
- very flexible
Operator Interface

Option 1: Full Materialization

Every operator materializes its output. The input is always read from a materialized state.

Advantages:
- easy to implement
- can handle surprises concerning intermediate result sizes (dynamic plans)
- advanced techniques like parallelization, result sharing, etc. are simple

Disadvantages:
- materialization is expensive
- in particular if data is larger than main memory

Few systems use this approach, but some do (MonetDB).
Operator Interface (2)
Option 2: Iterator Model

Each operator produces a tuple stream on demand. The input is iterated over.

Advantages:
- data is pipelined between operators
- avoids unnecessary materialization
- flexible control flow
- easy to implement

Disadvantages:
- millions of virtual function calls
- poor locality

The standard model. Widely used.
Operator Interface (3)

The iterator model usually offers the following interface:

- open
- next
- close

Repeated calls to next produce the output stream.

Internally, operators maintain a complex state to offer the iterator interface.
Operator Interface (4)

How to pass data from one operator to the other?

- the data itself is opaque
- as a consequence, it cannot be passed (easily) by value

Alternative 1: pass tuple pointers

- the real data resides on a page/in the buffer
- operators are only passed pointers to the data

Alternative 2: not at all

- there is a global data space ("registers")
- subscript functions operate on these registers
- the operators never touch the data directly

Alternative 2 is more generic, and can cope better with computed columns.
Operator Interface (5)

Option 3: blockwise processing
Each operator produces a tuple stream, but not tuple-by-tuple but as a stream of larger chunks.

Advantages:
- far fewer function calls
- better code and data locality

Disadvantages:
- additional materialization overhead
- consumes memory bandwidth
- control flow not as flexible
Operator Interface (6)

Option 4: pushing tuples up
Each operator pushes produced tuples towards the consuming operators.

Advantages:
- operator logic is concentrated in a few loops
- good code and data locality
- pipelining etc. still possible
- support for DAG-structured plans

Disadvantages:
- some restrictions in control flow
- code generation more involved
Examples - Full Materialization

scan($R$)
   // no-op, all operators read their input
   return $R$

select($R,p$)
   $R'$=new temporary relation
   for each $t \in R$
      if $p(t)$
         append $t$ to $R'$
   return $R'$

cross($R_1,R_2$)
   $R'$=new temporary relation
   for each $t_1 \in R_1$
      for each $t_2 \in R_2$
         append $t_1 \circ t_2$ to $R'$
   return $R'$
Examples - Iterator Model

class Scan
    in, tid, limit

Scan::open($R$)
    in = $R$
    tid = 0
    limit = $|R|$

Scan::next()
    if tid $\geq$ limit
        return false
    load tuple $t$ from $in$ at position tid
    tid = tid + 1
    return true
Examples - Iterator Model (2)

```cpp
class Select
    in, p

Select::open(in, p)
    this.in = in
    this.p = p

Select::next(in, p)
    while in.next()
        if p()
            return true
    return false
```
Examples - Iterator Model (3)

```cpp
class Cross

    left, right, step

Cross::open(left, right)

    this.left = left

    this.right = right

    step = true

Cross.next()

    while true

        if step

            if not left.next()

                return false

            right.open()

        step = false

        if right.next()

            return true

        step = true
```
Examples - Blockwise Processing

```cpp
class Scan
   in, tid, limit

Scan::open(R)
   in = R
   tid = 0
   limit = |R|

Scan::next()
   C = min(limit - tid, 1000)
   R' = tuple array of size C
   for i = 0...C - 1
      load tuple R'[i] from in at position tid + i
   tid = tid + C
   return R'
```
Examples - Blockwise Processing (2)

```java
class Select
    in, p
Select::open(in, p)
    this.in = in, this.p = p
Select::next(in, p)
    while true
        R' = in.next()
        if |R'| = 0
            return R'
        w = 0
        for i = 0...|R'| - 1
            R'[w] = R'[i]
            w = w + p(R'[w])
        R'.length = w
        if |R'| > 0
            return R'
```

Examples - Blockwise Processing (3)

```plaintext
class Cross
    left,right,c_L,l_L,R_L,c_R,l_R,R_R

Cross::open(left,right)
    this.left=left
    this.right=right
    step=true
    c_L = l_L = c_R = r_R = 0

Cross.next()
    R'=tuple array of size 1000, w=0
```
Examples - Blockwise Processing (4)

while true
    while $c_R = l_R$
        $c_L = c_L + 1$
        if $c_L \geq l_L$
            $R_L = \text{left}.\text{next}()$
            if $|R_L| = 0$
                $R'.\text{length}=w$, return $R'$
            $c_L = 0$, $l_L = |R_L|$
    $R_R = \text{right}.\text{next}()$
    if $|R_R| = 0$
        right.\text{rewind}()
        $c_R = 0$, $l_R = |R_R|$
    $R'[w] = R_L[c_L] \circ R_R[c_R]$
    $c_R = c_R + 1$, $w = w + 1$
    if $w = |R'|$
        return $R'$
Examples - Push

```cpp
class Scan
    consumer, R

Scan::open(consumer, R)
    this.consumer = consumer
    this.R = R

Scan::produce()
    for each t in R
        consumer.consume(t)
```
Examples - Push (2)

```cpp
class Select
    in, consumer, p

Select::open(in, consumer, p)
    this.in = in, this.consumer = consumer, this.p = p

Select::produce()
    in.produce()

Select::consume(t)
    if p(t)
        consumer.consume(p)
```
Examples - Push (3)

class Cross
  left, right, consumer, \( t_L \)

Cross::open(left, right, consumer)
  \texttt{this.left} = left, \texttt{this.right} = right, \texttt{this.consumer} = consumer

Cross::produce()
  left.produce()

Cross::consumeFromLeft(t)
  \( t_L = t \)
  right.produce()

Cross::consumeFromRight(t)
  consumer.consume(\( t_L \circ t \))
Additional Functionality

We ignored the \textit{close} function so far
- releases allocated resources

Other functionality implemented or used by operators:
- rewind/rebind
- memory management
- spooling intermediate results
Implementing Subscripts

The operators are query independent, but the subscripts are not
- cover the query-specific parts of the query
- attribute access (e.g., $x.a$)
- predicates (e.g., $a=b$)
- computations (e.g., $\text{sum}(\text{amount}*(1+\text{tax}))$)
- ...

Must be implemented, too
- different for every query
- but usually relatively simple
- complexity much lower than for operators
Implementing Subscripts (2)

Option 1: interpreter objects

Subscripts are assembled from interpreter objects.

- very flexible
- easy to implement
- widely used
- but: many virtual function calls

Val AccessInt::eval(char* ptr)

\[
\text{return } *((\text{int}*)(\text{ptr}+\text{ofs}));
\]

Val CompareEqInt::eval(char* ptr)

\[
\text{return } \text{left-}\rightarrow\text{eval(ptr).intValue}==\text{right-}\rightarrow\text{eval(ptr).intValue}
\]
Implementing Subscripts (3)

Option 2: virtual machines

Subscripts are compiled into instructions for a virtual machine.
- more efficient than interpreter objects
- but also more complex
- requires a compiler to byte code

```cpp
while (true) switch ((++op)->cmd) {
  case Cmd::AccessInt:
    reg[op->out]=*((*int)(ptr+op->val);
    break;
  case Cmd::CompareEqInt:
    break;
  ...
}
```
Implementing Subscripts (4)

Option 3: pre-compiled fragments

Subscripts are expressed as combination of pre-compiled fragments.
- each fragment performs a number of operations
- quite efficient (vectorization)
- but usually only applicable for column stores

```c
CompareEqInt(unsigned len, int* col1, int* col2, bool* result) {
    for (unsigned index = 0; index != len; ++index) {
        result[index] = col1[index] == col2[index];
    }
}
```
Implementing Subscripts (5)

Option 4: generated machine code

Subscripts are at runtime compiled into native machine code.
- the most efficient alternative
- but also the most difficulty
- portability is an issue
- we will look at this in the Section Code Generation

...  
movq  72(%rsp), %rax
movl  (%rax,%r12,4), %r13d
movq  120(%rsp), %rax
movl  (%rax,%r12,4), %edi
cmpl  %r13d,%edi
...
Pipelining

As mentioned, most approaches try to avoid copying data between operators

- this is called *pipelining*
- operators that do materialize their input are called *pipeline breakers*
- operators are consume their input completely before processing are called *full pipeline breakers*
- some binary operators are pipeline breakers on only one side

This behavior has implications regarding other operators.
Pipelining (2)

Some effects of different pipeline behavior

- if a pipeline break is between source and sink the original data is no longer accessible
  ▶ relevant for lazy attribute access/TID join/string representations etc.
  ▶ the system must plan defensively
- if a full pipeline breaker is between two operators both are decoupled
  ▶ the full pipeline break breaks the plan into fragments
  ▶ can be executed independent from each other
  ▶ relevant for scheduling
- ...

The code generation must know the pipeline behavior of operators.
Parallelization

How can we exploit multiple cores during query processing?

- inter-query parallelism is simple
- intra-query parallelism is much harder
- independent parts of the query can be executed in parallel (see: full pipeline breaker)
- parallelizing individual operators is more difficult
- usual strategy: partition the input

We will discuss this later in more detail.
Algebraic Operators
Algebraic Operators

Queries are translated into relational algebra

- therefore a DBMS must offer implementations for all algebraic operators
- often more than one implementation
- different implementations are tuned for different usage scenarios

Complexity varies

- a few operators are very simple
- but most are quite complex
- pipelining operators tend to be simple
- pipeline breakers tend to be complex
Table Scan

The most basic operator

- produces all tuples contained in a relation
- conceptually very simple
- implementation complexity varies from simple to complex
- has to navigate the physical representation of the relation
- additional complexity from deferred updates, snapshot isolation, etc.

TableScan::produce()

\[
\begin{align*}
\text{for each} & \quad \text{page extent } e \\
\text{for each} & \quad \text{page } p \text{ in } e \\
& \quad \text{fix } p \\
\text{for each} & \quad \text{tuple } t \text{ in } p \\
& \quad \text{consumer.consume}(t) \\
& \quad \text{unfix } p
\end{align*}
\]
Efficient Query Processing

Algebraic Operators

Selection

A selection $\sigma_p$

- filters out all tuples that do not satisfy $p$
- a very simple operator
- many systems do not even implement it as separate operator
- instead, piggybacked onto other operators

Select::produce()

\[
\text{input}.\text{produce()}
\]

Select::consume($t$)

\[
\text{if } p(t) \text{ then consumer.consume}(t)
\]
Map

A map $\chi_{a:f}$
- computes a new column by evaluating $f$
- another very simple operator
- like selections, often piggybacked

Map::produce()

\[ \text{input}.\text{produce()} \]

Map::consume($t$)

\begin{align*}
  t' &= t \circ [a : f(t)] \\
  \text{consumer}.\text{consume}(t')
\end{align*}
Join

A join $e_1 \bowtie_p e_2$
- a very complex operator
- one of the most important operators
- several different implementations exist

Candidate implementations depend on the join itself:
1. if $e_2$ depends upon $e_1$, nested loop join must be used (i.e. dependent join $e_1 \bowtie e_2$)
2. otherwise, if $p$ has the form $e_1.a = e_2.b$, any join algorithm can be used (equi-join)
3. otherwise, either nest loop or blockwise nested loop can be used
Nested-Loop Join

The nested loop join $\bowtie^\text{NL}_p$ is the most flexible, but also most simple and inefficient join

- evaluates the right hand side for every tuple of the left side
- pairwise comparison, suitable for any kind of predicate
- the right hand side is evaluated very frequently

\begin{align*}
\text{NLJoin::produce}() &= \\
left.\text{produce}() &= \\
\text{NLJoin::consumeFromLeft}(t) &= \\
&= t_L = t \\
&= right.\text{produce}() \\
\text{NLJoin::consumeFromRight}(t) &= \\
&= t' = t_L \circ t \\
&= \text{if } p(t') \\
&= \text{consumer}.\text{consume}(t')
\end{align*}
Blockwise-Nested-Loop Join

The blockwise nested loop join $\Join_{p}^{BNL}$

- loads as many tuples from the left side as possible
- evaluates the right side and joins
- and repeats this with additional chunks from the left side
- like a NL join, suitable for any predicate (but not for $\bowtie$)
- greatly reduces the number of passes over the right hand side
- potentially speeds up execution by orders of magnitude

BNLJoin::produce()
- clear the tuple buffer
  - left.produce()
  - if tuple buffer is not empty
    - right.produce()
Blockwise-Nested-Loop Join (2)

BNLJoin::consumeFromLeft(\(t\))

\textbf{if} not \textbf{can} materialize \(t\)

\textit{right.produce()}

clear the tuple buffer
materialize \(t\) in tuple buffer

BNLJoin::consumeFromRight(\(t\))

\textbf{for each} \(t_L\) \textbf{in} the tuple buffer

\(t' = t_L \circ t\)

\textbf{if} \(p(t')\)

\textit{consumer.consumer(t')}
Sort-Merge Join

The sort-merge join $e_1 \Join_p^{SM} e_2$

- assumes that $p$ has the form $e_1.a = e_2.b$, that $e_1$ is sorted on $a$, and that $e_2$ is sorted on $b$
- some implementations actually perform the sort, too, but we consider it as separate operator
- performs a linear pass over the input to find matching entries
- very fast and efficient (after sorting)
- **but:** only simple for the $1:N$ case!

Note: a SM join is simple in the iterator model, but not in the push model (control flow). We materialize the left hand side here to simplify the code, which is usually not needed.
Sort-Merge Join (2)

SMJoin::produce()
    prepare a temp segment for spooling
    left.produce()
    \( I_L = \text{first spooled tuple} \)
    right.produce()

SMJoin::consumeFromLeft(\( t \))
    spool \( t \) to temp segment
Sort-Merge Join (3)

$\text{SMJoin::consumeFromRight}(t)$

\[
\text{while } (*l_L).a < t.b \\
\quad \text{advance } l_L \\
\]

$\quad l_B = l_L$

\[
\text{while } (*l_B).a = t.b \\
\quad \text{consumer.consume}((*l_B) \circ t) \\
\quad \text{advance } l_B
\]

- non-standard, as the left hand side is already materialized
- one usually tries to avoid this
- better: parallel scans through both sides
- materialization only if an $n:m$ match is found
Hash-Join

The hash join $e_1 \Join^H e_2$

- assumes that $p$ has the form $e_1.a = e_2.b$
- builds a hash table from $e_1$, and probes the hash table with $e_2$
- in-memory case and external memory case
- real implementations offer both in one (first in-memory, external memory when needed)
- we split both cases to simplify the code

HJJoinInMem::produce()

  prepare an in-memory hash table
  left.produce()
  right.produce()
Hash-Join (2)

HJJoinInMem::consumeFromLeft(t)
    store t in hash table[t.a]

HJJoinInMem::consumeFromRight(t)
    for each \( t_L \) in hash table[t.b]
        if \( p(t_L \circ t) \)
            consumer.consume(t_L \circ t)

- for the combined case consumeFromLeft would check for overflows
- switch to external memory hash join when memory is full
Hash-Join (3)

HJJoinExternal::produce()
  prepare an in-memory spool buffer
  prepare a temp segment for spooling
  define initial partition boundaries
  \textit{left}.produce()
  flush the buffer if needed
  repartitionLeft()
  \textit{right}.produce()
  flush the buffer if needed
  joinPartitions()

- left side and right side are partitioned
- recursive re-partition might be needed
- once partitions fit, partitions can be join in-memory
Hash-Join (4)

HJJoinExternal::consumeFromLeft(t)

if not can spool t to the buffer
    sort the buffer by hash values
    write entries in corresponding partitions
    update partition statistics
    empty the buffer
spool t into buffer

- left side is materialized in memory
- when memory is full, data is written to corresponding partitions on disk
- here: sort to produce sequential I/O
Hash-Join (5)

HJJoinExternal::repartitionLeft()  
for each partition larger than main memory  
   if number of different hash values > 1  
      derive finer partition bounds within the partition  
      load partition piecewise into memory  
      write out each piece into finer partitions  
      replace large partition with finer partitions  
   else  
      mark the partition as overflow

- breaks large partitions into finer partitions
- until the partition fits into main memory
- overflow partitions are a special case
Hash-Join (6)

HJJoinExternal::consumeFromRight(t)

if not can spool t to the buffer
  sort the buffer by hash values
  write entries in corresponding partitions
  empty the buffer
spool t into buffer

- materializes, just like the left side
- but can use the proper partition boundaries now
Hash-Join (7)

HJJoinExternal::joinPartitions()

for each partition $P$
  if $P$ is an overflow partition
    joinPartitionOverflow($P$)
  else
    joinPartition($P$)

- partitions are joined pair-wise
- due to the equal join, join partners can only be found in corresponding partitions
- the overflow case needs some special care
Hash-Join (8)

HJJoinExternal::joinPartition(P)
load P for the left side into a hash table
for each t in the right partition P
  for each t_L in hash table[t.b]
    if p(t_L ◦ t)
      consumer.consume(t_L ◦ t)

- P is known to fit for the left side
- brings it to the in-memory case
Hash-Join (9)

HJJoinExternal::joinPartitionOverflow(P)

for each memory sized chunk of P from left
load the chunk into a hash table

for each t in the right partition P

for each $t_L$ in hash table[$t.b$]

if $p(t_L \circ t)$

consumer.consume($t_L \circ t$)

- partition did not fit (identical values)
- similar to a blockwise-nested-loop join
- but uses a hash-table to speed up finding matches
**Singleton Join**

A singleton-join is a join $e_1 \Join^1 J e_2$

- where $|e_1|$ is guaranteed to be $\leq 1$
- relatively common, e.g., after group-by, scalar subqueries, etc.
- nested loop join would work, of course
- but singleton is even simpler (see following slides)

SingletonJoin::produce()

```
left.produce()
right.produce()
```

SingletonJoin::consumeFromLeft($t$)

```
t_L = t
```

SingletonJoin::consumeFromRight($t$)

```
if $p(t_L \circ t')$
  consumer.consume($t_L \circ t$)
```
Non-Inner Joins

So far we have only consider inner joins. But:

- $\bowtie$, $\Join$, $\bowtie$ have to produce non-matching tuples, too
- $\bowtie$, $\Join$ have to produce only matching tuples, without multiplicity
- $\bowtie$, $\Join$ have to produce only non-matching tuples, without multiplicity

Similar mechanism, but requires additional bookkeeping.
Non-Inner Joins (2)

Idea: non-inner joins *mark* tuples with a join partner

- outer joins: additional pass, produce non-marker tuples with NULL values
- semi joins: produce only if not marked yet
- anti joins: additional pass, produce only non-marked tuples

Problem: where to store the marker bit?
Non-Inner Joins (3)

Marking the left side is usually simple:

- left tuples in-memory for potential join partners
- reserve a bit in the memory block/hash table/...
- bit can be examined before flushing the memory

Marking the right side is more problematic:

- ok if a right hand tuple is only considered once
- then, store the marker in a register
- but nested loop joins etc. are difficult
- maintain extra data structure (interval compression)
Sort

Sorting is useful for a number of other operations (sm-join, aggregation, duplicate elimination, etc.)

Basic strategy:

1. load chunks of tuples into memory
2. sort in-memory, write out sorted runs
3. merge sorted runs
4. recurse if needed to handle merge fanout

Implementation varies a bit.
Sort (2)

Sort::produce()
  input.produce()
  flush memory
  merge partitions
  for each \( t \) in merged partitions
    consumer.consume(\( t \))

Sort::consume(\( t \))
  if not enough memory to store \( t \)
    flush memory
    materialize \( t \) in memory

How to implement flush and merge?
Sort (3)

Easy solution for flushing:

- sort the in-memory tuples using quick sort
- write out all of them, release all memory
- fast sort algorithm, simple

More complex solution:

- use heap sort with replacement selection
- write out only when needed
- produces longer runs
- for sorted input: one run
- variable-sized records are difficult to handle
Efficient Query Processing  Algebraic Operators

Sort (4)

Merging is usually performed on the fly:

- read all runs in parallel
- retrieve always the smallest element
- tree of losers, or some other priority queue

Problem: what if the number of runs is too large?

- perform a partial merge
- reduces the number of runs
- repeat until merge is feasible
Group By

Aggregation can be implemented in two ways:

Sort based
- input is sorted by group-by attributes
- all tuples within one group are neighboring
- aggregation is largely trivial

Hash based
- aggregate into hash table
- spill to disk if needed
- merge spilled partitions, similar to hash join partitioning
Group By (2)

InMemoryGroupBy::produce()
   initialize an empty hash table
   input.produce()
   for each group $t$ in hash table
      consumer.consume($t$)

InMemoryGroupBy::consume($t$)
   if hash table[$t|A$] exists
      update the group hash table[$t|A$]
   else
      create a new group in hash table[$t|A$]

Variable-length aggregates require some care.
Set Operations

The DBMS also has to offer set operations

- union / union all
- intersect / intersect all
- except / except all

Somewhat similar to joins, but have a very specific behavior.
Set Operations (2)

union all

The only trivial set operations
- concatenates both tuple streams
- attribute rename required
- but otherwise nearly no code
Set Operations (3)

union

Like a union, but without duplicates

- can be implemented as `union all` followed by duplicate elimination
- \( e_1 \cup e_2 \equiv \Gamma_A(e_1 \cup e_2) \)
- or: directly write into one hash table
  (slightly more efficient, but nearly identical)
Set Operations (4)

**intersect**

Similar to a semi-join

- $e_1 \cap e_2 \approx e_1 \times e_2$
- only true if $e_1$ is duplicate free
- can be checked during the build phase
- or: use a strategy like for intersect all
Set Operations (5)

intersect all

Intersection with bag semantics

- defined via characteristics functions
- group by sides into one hash table
- count occurrences on left and right side
- intersect result is minimum of both
- standard group-by algorithm (including external memory etc.)
- for in-memory case can prune right side
Set Operations (6)

except

Similar to a anti-join

- \( e_1 \setminus e_2 \approx e_1 \dagger e_2 \)
- only true if \( e_1 \) is duplicate free
- can be checked during the build phase
- or: use a strategy like for except all
Set Operations (7)

except all

Set difference with bag semantics
- defined via characteristics functions
- group by sides into one hash table
- count occurrences on left and right side
- except result is $\max(0, l_c - r_c)$
- standard group-by algorithm (including external memory etc.)
- for in-memory case can prune right side
Code Generation
Motivation

For good performance, the operator subscripts have to be compiled
- either byte code
- or machine code
- generating machine code is more difficult but also more efficient

Machine code has portability problems
- code generation frameworks hide these
- some well known kits: LLVM, libjit, GNU lightning, ...
- greatly simplify code generation, often offer optimizations

LLVM is one of the more mature choices.
LLVM

Distinct characteristics

- unbounded number of registers
- SSA form
- strongly typed values

```c
define i32 @fak(i32 %x) {
    %1 = icmp ugt i32 %x, 1
    br i1 %1, label %L1, label %L2
L1:  %2 = sub i32 %x, 1
    %3 = call i32 @fak(i32 %2)
    %4 = mul i32 %x, %3
    br label %L3
L2:  br label %L3
L3:  %5 = phi i32 [ %4, %L1 ], [ 1, %L2 ]
    ret i32 %5
}
```
Compiling Scalar Expressions

- all scalar values are kept in LLVM registers
- additional register for NULL indicator if needed
- most scalar operations (eq, +, -, etc.) compile to a few LLVM instructions
- C++ code can be called for complex operations (like etc.)
- goal: minimize branching, minimize function calls

The real challenge is integrating these into set-oriented processing.
Data-Centric Query Execution

Why does the iterator model (and its variants) use the operator structure for execution?

- it is convenient, and feels natural
- the operator structure is there anyway
- but otherwise the operators only describe the data flow
- in particular operator boundaries are somewhat arbitrary

What we really want is **data centric** query execution

- data should be read/written as rarely as possible
- data should be kept in CPU registers as much as possible
- the code should center around the data, not the data move according to the code
- increase locality, reduce branching
Data-Centric Query Execution (2)

Example plan with visible pipeline boundaries:

- data is always taken out of a pipeline breaker and materialized into the next
- operators in between are passed through
- the relevant chunks are the pipeline fragments
- instead of iterating, we can push up the pipeline
Data-Centric Query Execution (3)
Corresponding code fragments:

initialize memory of $\Join_{a=b}$, $\Join_{c=z}$, and $\Gamma_z$

for each tuple $t$ in $R_1$
  if $t.x = 7$
    materialize $t$ in hash table of $\Join_{a=b}$

for each tuple $t$ in $R_2$
  if $t.y = 3$
    aggregate $t$ in hash table of $\Gamma_z$

for each tuple $t_3$ in $R_3$
  for each match $t_2$ in $\Join_{z=c}[t_3.c]$
    for each match $t_1$ in $\Join_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$
Data-Centric Query Execution (4)

Basic strategy:

1. the producing operator loops over all materialized tuples
2. the current tuple is loaded into CPU registers
3. all pipelining ancestor operators are applied
4. the tuple is materialized into the next pipeline breaker

- tries to maximize code and data locality
- a tight loops performs a number of operations
- memory access in minimized
- operator boundaries are blurred
- code centers on the data, not the operators
Producing the Code

Code generator mimics the produce/consume interface

- these methods do not really exist, they are conceptual constructs
- the *produce* logic generates the code to produce output tuples
- the *consume* logic generates the code to accept incoming tuples
- not clearly visible within the generated code
Producing the Code (2)

```cpp
void HJTranslatorInner::produce(CodeGen& codegen, Context& context) const
{
    // Construct functions that will be called from the C++ code
    {
        AddRequired addRequired(context, getCondition().getUsed().limitTo(left));
        produceLeft = codegen.derivePlanFunction(left, context);
    }
    {
        AddRequired addRequired(context, getCondition().getUsed().limitTo(right));
        produceRight = codegen.derivePlanFunction(right, context);
    }

    // Call the C++ code
    codegen.call(HashJoinInnerProxy::produce.getFunction(codegen),
                    {context.getOperator(this)});
}

void HJTranslatorInner::consume(CodeGen& codegen, Context& context) const
{
    llvm::Value ∗opPtr = context.getOperator(this);
```
Producing the Code (3)

// Left side
if (source==left) {
  // Collect registers from the left side
  vector<ResultValue> materializedValues;
  matHelperLeft.collectValues(codegen, context, materializedValues);

  // Compute size and hash value
  llvm::Value* size=matHelperLeft.computeSize(codegen, materializedValues);
  llvm::Value* hash=matHelperLeft.computeHash(codegen, materializedValues);

  // Materialize in hash table, spools to disk if needed
  llvm::Value* ptr=codegen.callBase(HashJoinProxy::storeLeftInputTuple,
      {opPtr, size, hash});
  matHelperLeft.materialize(codegen, ptr, materializedValues);
Efficient Query Processing

Code Generation

Producing the Code (4)

// Right side
} else {
  // Collect registers from the right side
  vector<ResultValue> materializedValues;
  matHelperRight.collectValues(codegen, context, materializedValues);

  // Compute size and hash value
  llvm::Value* size = matHelperRight.computeSize(codegen, materializedValues);
  llvm::Value* hash = matHelperRight.computeHash(codegen, materializedValues);

  // Materialize in memory, spools to disk if needed, implicitly joins
  llvm::Value* ptr = codegen.callBase(HashJoinProxy::storeRightInputTuple,
                                        {opPtr, size});
  matHelperRight.materialize(codegen, ptr, materializedValues);
  codegen.call(HashJoinInnerProxy::storeRightInputTupleDone, {opPtr, hash});}
void HJTranslatorInner::join(CodeGen& codegen, Context& context) const
{
    llvm::Value* leftPtr = context.getLeftTuple(), *rightPtr = context.getLeftTuple();
    // Load into registers. Actual load may be delayed by optimizer
    vector<ResultValue> leftValues, rightValues;
    matHelperLeft.dematerialize(codegen, leftPtr, leftValues, context);
    matHelperRight.dematerialize(codegen, rightPtr, rightValues, context);

    // Check the join condition, return false for mismatches
    llvm::BasicBlock* returnFalseBB = constructReturnFalseBB(codegen);
    MaterializationHelper::testValues(codegen, leftValues, rightValues,
        joinPredicateIs, returnFalseBB);

    for (auto iter = residuals.begin(), limit = residuals.end(); iter != limit; ++iter) {
        ResultValue v = codegen.deriveValue(**iter, context);
        CodeGen::If checkCondition(codegen, v, 0, returnFalseBB);
    }

    // Found a match, propagate up
    getParent() -> consume(codegen, context);
}
Parallel Query Execution
Parallelism

Why parallelism

- multiple users at the same time
- modern server CPUs have dozens of CPU cores
- better utilize high-performance IO devices

Forms of parallelism

- inter-query parallelism: execute multiple queries concurrently
  - map each query to one process/thread
  - concurrency control mechanism isolates the queries
  - except for synchronization that parallelism is “for free”
- intra-query parallelism: parallelize a single query
  - horizontal (bushy) parallelism: execute independent sub plans in parallel (not very useful)
  - vertical parallelism: parallelize operators themselves
Vertical Parallelism: Exchange Operator

- optimizer statically determines at query compile-time how many threads should run
- instantiates one query operator plan for each thread
- connects these with “exchange” operators, which encapsulate parallelism, start threads, and buffer data
- relational operator can remain (largely) unchanged
- often (also) used in a distributed setting

```
Xchg(3:1)
```

```
R
  r
  |  Xchg(3:1) |
  r         |
  |          |
  v         |
  |          |
R          |
  r         |
  |          |
  v         |
  |          |
R1         |
  r         |
  |          |
  v         |
  |          |
R2         |
  r         |
  |          |
  v         |
  |          |
R3         |
```
Exchange Operator Variants

- Xchg(N:M) N input pipelines, M output pipelines

Many useful variants

- XchgUnion(N:1) specialization of Xchg
- XchgDynamicSplit(1:M) specialization of Xchg
- XchgHashSplit(N:M) split by hash values
- XchgBroadcast(N:M) send full input to all consumers
- XchgRangeSplit(N:M) partition by data ranges
Aggregation with Exchange Operators (3-way parallelism)
Join with Exchange Operators (3-way parallelism)
Disadvantages of Exchange Operators

- static work partitioning can cause load imbalances (large problem with many threads)
- degree of parallelism cannot easily be changed mid-query (workload changes)
- overhead:
  - usually implemented using more threads than CPU cores (context switching)
  - hash re-partitioning often does not pay off
  - exchange operators create additional copies of the tuples
Parallel Query Engine

- alternative to Exchange Operators: parallelize operators themselves
- requires synchronization of shared data structures (e.g., hash tables)
- allows for more flexibility in designing parallel algorithms for relational operators
Morsel-Driven Query Execution

- break input into constant-sized work units ("morsels")
- dispatcher assigns morsels to worker threads
- \# worker threads = \# hardware threads
Dynamic Scheduling

- the total runtime of a query is the runtime of the slowest thread/core/machine
- when dozens of cores are used, often a single straggler is much slower than the others (e.g., due to other processes in the system or non-uniform data distributions)
- solution: don’t partition input data at the beginning, but use dynamic work stealing:
  - synchronized queue of small jobs
  - threads grab work from queue
  - the `parallel_for` construct can provide a high-level interface
Parallel In-Memory Hash Join

1. build phase:
   1.1 each thread scans part of the input and materializes the tuple
   1.2 create table of pointers of appropriate size (tuple count sum of all threads)
   1.3 scan materialized input and add pointers from array to materialized tuples using atomic instructions

2. probe phase: can probe the hash table in parallel without any synchronization (as long no marker is needed)
Main-Memory Databases
Motivation

Hardware trends

- Huge main memory capacity
  with complex access characteristics (Caches, NUMA)
- Many-core CPUs
- SIMD support in CPUs
- New CPU features (HTM)
- Also: Graphic cards, FPGAs, low latency networking, . . .

Database system trends

- Entire database fits into main memory
- New types of database systems
- New algorithms, new data structures

“The End of an Architectural Era.
(It’s Time for a Complete Rewrite).”
Recap: Database Workloads

Analytics
- Long-running
- Access large parts of the database
- Often use scans
- Read-only
- Example: “Average order value per year and product group?”

Transaction processing
- Short running
- (Multiple) point queries + simple control flow
- Insert/Update/Delete/Read data
- Example: “Increment account x by 10, decrement account y by 10”

Universal DBMS used for both (but not concurrently).
OLTP

Universal DBMS were optimized for 1970’s hardware

- Small fraction of DB in memory buffer
- Hide and avoid disk access at any cost

Today

- Even enterprises can store entire DB in memory
- Transaction are often “one-shot”
- Transactions execute in a few ms or even µs
OLTP (2)

Main sources of overhead

- ARIES-style logging
- Locking (2PL)
- Latching
- Buffer Management

Useful work can be as low as $\frac{1}{60}$th of instructions\(^1\). Modern systems avoid this overhead (see slide 360).

\(^1\)Harizopoulos et al. – *OLTP Through the Looking Glass, and What We Found There*
Physical Data Layout in Main Memory

Lightweight:
- Buffer Manager removed
- No need for segments
- No need for slotted pages

Store data in simple arrays. But: Row-wise or column-wise?

Logical Table

<table>
<thead>
<tr>
<th>EmpId</th>
<th>First</th>
<th>Last</th>
<th>Salery</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Joe</td>
<td>Doe</td>
<td>40000</td>
</tr>
<tr>
<td>20</td>
<td>Mary</td>
<td>Jones</td>
<td>32000</td>
</tr>
<tr>
<td>32</td>
<td>Bob</td>
<td>Black</td>
<td>60000</td>
</tr>
<tr>
<td>25</td>
<td>Jane</td>
<td>Jones</td>
<td>85000</td>
</tr>
</tbody>
</table>
Physical Data Layout in Main Memory (2)

Row Store:
- Beneficial when accessing many attributes
- For OLTP

Column Store:
- Excellent cache utilization
- Sometimes individually sorted
- Compression potential
- Vectorized processing
- For OLAP

Hybrid Row/Column Stores possible
New Systems (Examples)

OLTP-only:
- VoltDB/H-Store
- Microsoft Hekaton

OLAP-only:
- Vectorwise
- MonetDB
- DB2 BLU

Hybrid OLTP and OLAP:
- SAP HANA
- HyPer
New Systems: OLTP (Examples)

Challenge:

- Avoid overhead
- Guarantee ACID

Approaches:

- Buffer Management: Removed
- Logging
  - H-Store/VoltDB: Log shipping to other nodes
  - Hekaton: Lightweight logging (no index structures)
- Locking:
  - H-Store/VoltDB: Serial execution (on private partitions)
  - Hekaton: Optimistic MVCC
- Latching
  - H-Store/VoltDB: Not necessary
  - Hekaton: Latch-free data structures
New Systems: Hekaton

- Integrated in SQL Server
- Code Generation
- Only access path: Index (Hash or B(w)-Tree)
- Latch-Free Indexes
- MVCC
New Systems: OLAP

- Vectorwise: Vectorized Processing
- HyPer: Query Compilation (cf. Chapter *Code Generation*)
New Systems: Hybrid OLTP and OLAP

Traditionally:
- Mixing OLTP and OLAP leads to performance decline
- ETL architecture
- 2 systems, stale data

New Systems
- SAP HANA
  - Split DB into read-optimized *main* and update-friendly *delta*
  - OLAP queries read main, OLTP transactions read delta *and* main
  - Periodically merge main and delta
- HyPer: Virtual memory snapshots
HyPer: Virtual Memory Snapshots

OLTP Data

OLTP Tx

A
B
C
D
E
F
G
H
HyPer: Virtual Memory Snapshots

OLTP Data

forked OLAP-Snapshot

OLAP Queries

OLTP Tx

A
B
C
D
E
F
G
H
HyPer: Virtual Memory Snapshots

![Diagram of HyPer](image)

- OLTP Data
- forked OLAP-Snapshot
- OLAP Queries
- OLTP Tx
- read C
HyPer: Virtual Memory Snapshots

OLTP Data

forked OLAP-Snapshot

OLAP Queries

update $C$ to $C^*$

copy-on-write

OLTP Tx

update $C$ to $C^*$

copy-on-write
HyPer: Virtual Memory Snapshots

forked OLAP-Snapshot

OLTP Data

OLAP Queries

read C

read H

OLTP Tx

C
D
A
B
G
H
E
F

C*

OLTP Data

OLAP Queries
In-Memory Index Structures

- In-memory hash indexes
  - Simple and fast
  - Growing is very expensive
  - Do not support range queries

- Search Trees
  - BSTs are cache unfriendly
  - B-Trees better (even though designed for disk)

- Radix-Trees ("Tries")
  - Support range queries
  - Height is independent from number of entries
Radix Trees

Properties:
- Height depends on key length, not number of entries
- No rebalancing
- All insertion orders yield same tree
- Keys are stored in the tree implicitly

Search:
- Node is array of size $2^s$
- $s$ bits (often 8) are used as an index into the array
- $s$ is a trade-off between lookup-performance and memory consumption
Adaptive Radix Trees

Four node types:

- **Node4**: 4 keys and 4 pointers at corresponding positions:

  ![Node4 Diagram]

- **Node16**: Like Node4, but with 16 keys. SIMD searchable.

- **Node48**: Full 256 keys (index offset), point to up to 48 values:

  ![Node48 Diagram]

- **Node256**: Regular trie node, i.e. array of size 256

  Additionally: Header with node type, number of entries
Exploiting HTM for OLTP

- Intel’s Haswell introduced HTM (via cache coherency protocol)
- Allows to group instructions to transactions
- Can help to implement DB transactions, but
  - Do not guarantee ACID by themselves
  - Limited in size/time

⇒ Use HTM transactions as building blocks for DB transactions
Exploiting HTM for OLTP (2)

Goals:
- As fine-grained as 2PL, but faster
- As fast as serial execution, but more flexible

atomic-elide-lock (lock) {
    account[from] -= amount;
    account[to] += amount;
}
Implementing DB transactions with HTM

Use TSO + HTM for latching:

- Database transaction
  - Conflict detection: read/write sets via timestamps
  - Elided lock: serial execution

- Request a new timestamp, record safe timestamp

- HTM transaction
  - Conflict detection: read/write sets in hardware
  - Elided lock: latch
  - Single tuple access
  - Verify/update tuple timestamps

- Release timestamp, update safe timestamp

- Relation and index structure layout must avoid conflicts
NUMA-Aware Data Processing

NUMA architectures:

- **Nehalem EX**
  - Socket 0:
    - 8 cores
    - 24MB L3
  - Socket 1:
    - 8 cores
    - 24MB L3
  - Socket 3:
    - 8 cores
    - 24MB L3
  - Bandwidth: 25.6GB/s

- **Sandy Bridge EP**
  - Socket 0:
    - 8 cores
    - 20MB L3
  - Socket 1:
    - 8 cores
    - 20MB L3
  - Socket 3:
    - 8 cores
    - 20MB L3
  - Bandwidth: 51.2GB/s

- **Local access cheap**
- **Remote access expensive**
NUMA-Aware Data Processing: Hash Join

**Phase 1**: process T morsel-wise and store NUMA-local storage area

**Phase 2**: scan NUMA-local storage area and insert pointers into HT

- Storage area of red core
- Storage area of green core
- Storage area of blue core

Insert the pointer into HT

...\(T\) ...\(T\) ...\(T\)

...\(R\) ...\(R\) ...\(R\)
Compaction

- OLTP & OLAP share the same physical data model
  - Fast modifications vs scan performance
  - Row store vs column store
- Modifications require snapshot maintenance
  - Use more memory
  - Congest memory bus
  - Stall transactions
Compaction: Hot/Cold Clustering

- Compression is applied asynchronously to cold part:
  - Dictionary encoding
  - Run-length encoding
  - Other schemes possible
- Compact snapshots through a mix of regular and huge pages
  - Keeps page table small
  - Clustered updates
  - No huge pages need to be replicated
Compaction: Hot/Cold Clustering

Cooling
- Hot & cold items mixed
- Uncompressed
- Small memory pages

Frozen
- Cold & compressed data
- Huge memory pages
- Rarely accessed by OLTP
- Immutable: Deleted and updated items are marked "invalid" and copied to Hot

Hot
- Working Set (hot data)
- Uncompressed
- Small memory pages

Small memory page

Cold
- Cold data items only
- Not compressed yet

"Invalid frozen items" data structure

Huge memory page
Compaction: Hot/Cold Clustering

How to detect temperature without causing overhead?

1. Software: LRU lists, counters
2. Hardware: mprotect
3. Hardware: dirty and young flags
Data Blocks

- most data is cold and rarely / never changes
- it is attractive to compress these aggressively
- and pre-compute SMAs
- helps with skipping data
- fits well with a cloud storage setup
Data Blocks - Scan Types

- **uncompressed chunk**: Vectors of e.g., 8192 records
- **interpreted vectorized scan on uncompressed chunk**
- **JIT-compiled tuple-at-a-time scan on uncompressed chunk**
- **hot scan**: Evaluation of scan predicates
- **single interface**: Tuple-at-a-time query pipeline
- **push matches tuple-at-a-time**
- **vectorized evaluation of SARGable predicates and copying of matches**
- **interpreted vectorized scan on Data Block**
- **cold scan**: Tuple
- **JIT-compiled tuple-at-a-time query pipeline**
- **PSMAs**
- **compressed Data Block**: Vectorized evaluation of SARGable predicates on compressed data and unpacking of matches
## Data Blocks - Layout

<table>
<thead>
<tr>
<th>tuple count</th>
<th>sma offset_0</th>
<th>dict offset_0</th>
<th>data offset_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression_0</td>
<td>string offset_0</td>
<td>sma offset_1</td>
<td>dict offset_1</td>
</tr>
<tr>
<td>data offset_1</td>
<td>compression_1</td>
<td>string offset_1</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>sma offset_n</td>
<td>dict offset_n</td>
<td>data offset_n</td>
</tr>
<tr>
<td>compression_n</td>
<td>string offset_n</td>
<td>min_0</td>
<td>max_0</td>
</tr>
</tbody>
</table>

### lookup table_0

#### Positional SMA index for attribute 0

<table>
<thead>
<tr>
<th>domain size_0</th>
<th>dictionary_0</th>
</tr>
</thead>
</table>

### compressed data_0

| string data_0 | min_1 | max_1 | ... |
Data Blocks - Vectorized Evaluation

- aligned data
- unaligned data
- read offset
- predicate evaluation
- movemask
- remaining data
- data
- precomputed positions table
- lookup
- add global scan position and update match vector
- write offset
- match positions

Data
read offset
aligned data
unaligned data
11
remaining data
predicate
evaluation
movemask
=15410
precomputed positions
1, 3, 5, 7, 9, 11, 14, 15, 17
+11
add global scan position and update match vector
write offset