Advanced Databases

Stratis D. Viglas

University of Edinburgh
Course logistics

- **Lecturer:** Stratis Viglas
  - email: sviglas@inf.ed.ac.uk
- **Days/Times:** Mon & Thu, 11:10-12:00
- **Office hours:** Mon, Thu 12:00-13:00 (or, by appointment)
  - Room: IF, 5.11
- **Course webpage:** [www.inf.ed.ac.uk/teaching/courses/adbs](http://www.inf.ed.ac.uk/teaching/courses/adbs)
- **Mailing list:** adbs-students@inf.ed.ac.uk
Syllabus

- Introduction
- Relational databases *overview*
  - *Data* model, *evaluation* model
- *Storage*
  - *Indexes*, *multidimensional* data
- Query *evaluation*
  - *Join* evaluation *algorithms*, *execution* models
- Query *optimisation*
  - *Cost* models, search space *exploration*, *randomised* optimisation
- *Concurrency* control and *recovery*
  - *Locking* and *transaction* processing
- *Parallel* databases
Assignments and software

- **Programming** assignments
- The *attica* database system
  - Home-grown *RDBMS*, written in Java
  - Visit inf.ed.ac.uk/teaching/courses/adbs/attica to download the system and the API documentation
  - All programming assignments will be using the *attica* front-end and code-base
- **Plagiarism policy**: You cheat, you’re caught, you fail
  - No discussion
Query cycle

- **Catalog**
- **Syntax tree**
- **Query Analyzer**
  - $\sigma R.a=5$
  - $\land R.a=T.b$
  - $\pi T.b$
- **Statistics**
  - **Optimiser**
    - **Relational algebra**
      - **Compilation**
        - Thread-id=12, operator=3, in-stream=1,...
      - **Execution environment**
        - **Scheduler**
          - **Execution plan**
            - nested-joins ($R, T,$ $a=b$)
            - hash-join ($T, S, b=c$)
            - index-scan ($T.b$)
            - index-scan ($S.c$)

- **Query Engine**
- **DB Tables**
- **Compiled code**
- **Results**
Three basic building blocks

- **Attribute**
  - A (name, value) pair

- **Tuple**
  - A set of attributes

- **Relation**
  - A set of tuples with the same schema
Data manipulation

- Operations to *isolate* a *subset* of a *single relation*: Selection (\(\sigma\)), Projection (\(\pi\))

- All *set operations*: Intersection, union, Cartesian product, set difference

- More *complex* operations: *Joins* (\(\bowtie\)), semi-joins, ...
Data storage

- **Disk drives** are organised in **records** of **512 bytes**
- The DB (and the OS) **I/O unit** is a **disk page** (typically, 4,096 bytes long)
- **Pages** (and records) are **stored** on **tracks**
- **Tracks** make up a **platter** (or a disk)
- **Platters** make up a **drive**
- The **same tracks** across all **platters** make up a **cylinder**
- The **disk head** (arm) reads the **same block** of **all tracks** on **all platters**
A bit of perspective

- The *dimensions* of the *head* are *impressive*\(^1\). With a *width* of less than a *hundred nanometers* and a *thickness* of about *ten*, it flies above the platter at a *speed* of up to *15,000 RPM*, at a *height* that is the equivalent of *40 atoms*. If you start multiplying these infinitesimally small numbers, you begin to get an idea of their significance.

- Consider this little *comparison*: if the *read/write head* were a *Boeing 747*, and the *hard-disk platter* were the *surface of the Earth*
  - The *head* would *fly* at *Mach 800*
  - At less than *one centimeter* from the *ground*
  - And *count* *every blade of grass*
  - Making *fewer than 10* unrecoverable counting *errors* in an *area* equivalent to all of *Ireland*

---

1Source: Matthieu Lamelot, Tom’s Hardware.
What about flash memory and solid state?

- The *geometry* is different
  - There are no tracks, or platters, or cylinders or anything of the sort
- But the *issues* are *similar*
  - Data is still accessed in *blocks*
  - Blocks are still organised in *pages*
  - *Sequential vs. random* I/O is still a *problem*
- Most of the things we say in this course are *applicable* to solid state as well
  - Added *complexity*: *write/read asymmetry*
Storing tuples

- Every \textit{disk block} contains
  - A \textit{header}
  - \textit{Data} (\textit{i.e.}, tuples)
  - \textit{Padding} (maybe)

- \textit{Two ways} of storing tuples
  - Either \textit{interleave tuples} of multiple relations, or
  - Keep the tuples of the \textit{same relation clustered}
Advantages of clustering

- **Scan** a relation of $X$ tuples, $Y$ tuples per block
  - If **unclustered**, worst case scenario: read $X$ blocks
  - **Clustered**: read $X/Y$ blocks

- How about **clustering** disk blocks?
  - **Reduces** unnecessary arm movement
The buffer manager

- Though the *data* is on *disk*, real *processing* is in *main memory*
- Disk blocks are read and put into the *buffer pool*
  - A collection of *memory pages*
- The *buffer manager* manages the buffer pool
  - Keeping track of *page references, replacing pages* if full, ...
What does the buffer manager do?

- When a *page is requested* it:
  - Checks to see *if the page is in* the buffer pool; if so it *returns it*
  - If not, it *checks whether there is room* in the buffer pool; if so it *reads it in and places it in the available room*
  - If not, it *picks a page for replacement*; if the page has been “touched” it *writes the page to disk and replaces it*
  - In all three cases, it *updates the reference count* for the requested page
  - If necessary, it *pins the new page*
  - It *returns* a *handle to the new page*
Page replacement

- **Least recently used** (LRU): check the number of references for each page; replace a page from the group with the lowest count (usually implemented with a priority queue)
  - Variant: *clock replacement*
- **First In First Out** (FIFO)
- **Most recently used** (MRU): the inverse of LRU
- **Random**!
Why not use the OS

- The OS implements virtual memory, so why not use it?
  - *Page reference patterns* and *pre-fetching*: the RDBMS in most cases *knows which page will be accessed later* (think of a clustered sequential scan)
  - Different *page replacement policies* according to the *reference pattern* (check p. 322 of your book)
  - *Page pinning*: certain *pages should not be replaced*
  - *Control* over *when a page is written to disk*: at times, pages need to be *forced to disk* (we’ll revisit that when discussing crash recovery)
Indexing and sorting

- Can be summarised as:
  - *Forget whatever you’ve learned about indexing, searching and sorting in main memory* (well, almost …)
- Remember, we are *operating over disk files*
  - The main idea is to *minimise disk I/O and not number of comparisons* (*i.e.*, complexity)
  - Just an idea: *comparing two values* in *memory* costs $4.91 \cdot 10^{-8}$ seconds; Comparing two values on *disk* costs $18.2 \cdot 10^{-5}$ seconds (3 orders of magnitude more expensive.)
Indexing functionality

- Indexes can be used for:
  - *Lookup* queries (e.g., \([\ldots]\text{ where value = ‘‘foo’’}\))
  - *Range* queries (e.g., \([\ldots]\text{ where value between 20 and 45}\))
  - *Join processing* (after all, predicates are value-based, aren't they?)

- The above uses, and much more, are what we call *access methods*
Two main classes

- **Tree-structured** indexes
  - Much like you would use a binary tree to search, but with a higher key-per-node cardinality
  - Retains order
  - Great for range queries
  - Both one-dimensional and multi-dimensional

- **Hash-based** indexes
  - Fully randomized (i.e., no order)
  - Great for single lookup queries
Sorted indexes

- The basic idea:
  - An index is on an (collection of) attribute(s) of a relation (called the index key)
  - It is much smaller than the relation
  - Index pages contain (key, pointer) pairs
    - key of the index
    - pointer to the data page
  - Plus one additional pointer (low key)
How does it answer range queries?

- Query is \( \text{low} \leq \text{value} \leq \text{high} \)
- Do a \textit{binary search} on the \textit{index file} to identify the \textit{page containing the low key}
- \textit{Keep scanning} the data file until the \textit{high key} is \textit{found}
- All \textit{done}!
Potential problem (and the solution)

- The *index* is *much smaller* than the *relation*, but it’s *still big*
- *Binary search* on it is *still expensive*
  - Remember, *data* is *on disk*
  - Have to access *half the index file pages, plus the pages satisfying the predicate*, all doing *random I/O*
- Why not build an *index on the index*?
  - *Tree!*
**B+ trees: the most widely used indexes**

- **Insertion/deletion** at $\log_f N$ cost ($f = \text{fanout}$, $N = \# \text{leaf pages}$)
- Tree is *height-balanced*
- Minimum *50% occupancy* (except for root)
- Characterised by its *order* $d$; *each node* contains $d \leq m \leq 2d$ entries
- *Equality* and *range* searches are efficient
B-tree example
B+ trees in practice

- Typical **order**: 100, typical **fill-factor**: 67%
  - Average **fan-out**: 133
- Typical capacities
  - Height 3: 2,532,637
  - Height 4: 312,900,700 (!)
- The **top levels** can often be kept **in memory**
  - 1st level: 4,096, or 8,192 bytes (1 page)
  - 2nd level: 0.5, or 1MB (133 pages)
  - 3rd level: 62, or 133MB
B+tree insertion

- **Find** correct leaf $L$
- **Put** data entry into $L$
  - If there is *enough space* in $L$, done!
  - If there is *no space*, $L$ needs to be *split* into $L$ and $L'$
  - *Redistribute* entries evenly in $L$ and $L'$
  - *Insert index entry* pointing to $L'$ into the *parent of $L$*

- **Ascend** the tree *recursively*, *splitting* and *redistributing* as needed

- **Tree tries to grow horizontally**; worst case scenario: a *root split* increases the height of the tree
B+-tree insertion: 8*

no room so it has to be split
B+tree insertion: 8*

Stratis D. Viglas (University of Edinburgh)
Insertion observations

- **Minimum occupancy** is guaranteed at both leaf and non-leaf pages.
- A *leaf split* leads to *copying* the key; a *non-leaf* split leads into *pushing up* the key (*why?*

The tree tries to *first grow horizontally* and if this is not possible, *then vertically*:

- In the example we could have *avoided* the *extra level* by *redistributing*
- But *in practice* this is *hardly ever done* (*why?*)
B+tree deletion

- **Find leaf L** where entry belongs
  - **Remove** the entry
  - If *L is half-full*, **done**!
  - If *L only has* $d - 1$ *entries*
    - Try to **redistribute** entries, **borrowing** from an **adjacent sibling** of *L*
    - If **redistribution fails**, **merge** *L* and its **sibling**
    - If **merge has occurred**, **delete** the **entry** for the **merged page** from the **parent of L**
- **Ascend** the tree **recursively**, performing the same algorithm
- **Merge** could **propagate to the root**, **decreasing** the trees **height**
B+tree deletion: 19*
B-tree deletion: 20*

If 20 is deleted, the minimum occupancy is compromised, so it must be redistributed. The middle key has been copied up (why?).
B+tree deletion: 24*

- Can't redistribute; must merge.
  - Middle key has been copied up (why?)
  - Which will result in an index merge.
  - This entry tossed out.

- No room so it has to be split.

- Can't redistribute; must merge.
B⁺-tree after deletion of 24*
Summary of B+-tree indexes

- **Ideal** for range searches, **good** for equality searches
- Highly **dynamic** structure
  - Insertions and deletions leave tree height-balanced, $\log f N$ cost
  - For most typical implementations, height is rarely greater than 3 or 4, occupancy at 67%
  - Which means that the index is almost always in memory! (remember the buffer pool?)
  - Almost always better than maintaining a sorted file
  - The most optimised RDBMS structure
Hash indexes

- **Hash-based indexes** are *good* for *equality* selections, *not* for *range* selections
  - In fact, they *cannot support range* selections (why?)
- **Static** and **dynamic techniques** exist here as well
  - *Trade-offs* similar to those between ISAM and B+trees
Static hashing

- **Number of primary pages fixed**
  - Allocated sequentially, never de-allocated
  - *Overflow pages* if needed
- \( h(k) \mod M = \text{bucket} \) to which *data entry* with *key* \( k \) belongs (\( M = \text{number of buckets} \))
Storage and indexing  One-dimensional indexing

Static hashing observations

- The *buckets contain* the *actual data!*
  - But *only* the *key* is *hashed*
  - *No secondary index* like in the tree case
- The *hash function must uniformly distribute* the *keys* across all buckets
  - Lots of ways to *tune* the hash function
- Again, *long overflow chains* of pages will develop, and pretty soon we’re doing *random I/O*
  - *Need* a *dynamic* technique (big surprise here…)
  - *Extendible hashing* to the rescue
Extendible hashing

- **Problem:** bucket (i.e., primary page) becomes full
- **Solution:** re-organize the file by doubling the number of buckets
  - Are you crazy? Reading and writing out everything is expensive!
  - Why not keep a directory of buckets and double only the directory?
    - Only read the bucket that overflowed
  - Directory much smaller; operation much cheaper
Extendible hashing example

- **Directory:** array of size 4
- **Key** $k$, apply **hash function** $h(k)$ and **translate** the result to **binary**
  - e.g., $h(k) = 5 = 101$
- **Last global depth number of bits** identify the **bucket**
Global, local depth and doubling

- **Global depth** (pertains to directory): maximum number of bits needed to tell which bucket an entry belongs to
- **Local depth** (pertains to bucket): maximum number of bits needed to tell whether an entry belongs to this bucket
- **Before** insertion (local = global) holds; if insertion causes (local > global) then directory needs to be doubled
Insertion example: $h(k) = 20$

20 = 10100

bucket must be split

these two buckets are called "split images"

but we now need 3 bits

4 = 00100
12 = 01100
32 = 10000
16 = 01000
20 = 10100

so we must double the directory
Doubling the directory

these originated from the same split bucket
Extendible hashing observations

- **Directory fits in memory**: equality search answered with only one disk I/O (two in the worst case!)
  - 100MB file, 100 bytes/tuple, 4kB pages, 1,000,000 data entries, 25,000 directory entries: fits in memory!
  - If the value distribution is skewed, directory grows large
  - Same hash-value entries are a problem (why?)

- **Deletion**: if removal empties bucket, then it can be merged with split image; if each directory entry points to the same bucket as its split image, the directory is halved
Linear hashing

- **Extendible hashing directory**: even if it is small, it is still a materialised level of indirection
- Though the number of buckets grows linearly, the size of the directory grows exponentially
- Objective: no directory, linear growth
- *Linear* hashing gets the job done
Why one, when you can have many?

- Key idea: instead of having a single hash function and using a set of bits, have multiple hash functions
  - Multiple hash functions implement the progressive doubling of the directory
- Allocate buckets not when they become full, but whenever we reach some predetermined load factor
- Single bucket allocation
- Each bucket allocation results in another hash function to be used
- Keep track of the number of buckets and the number of times the number of buckets has doubled
- Discard unused hash functions
In more detail

- Use a *family of hash functions* $h_0, h_1, h_2, \ldots$
  - $h_i(key) = g(key) \mod (2^i M)$
  - $M = \text{initial number of buckets}$
  - $g$ is some *hash function* (*range is not* $[0, \ldots, N - 1]$)
  - If $M = 2^{d_0}$, for some $d_0$, $h_i$ consists of *applying* $g$ and *looking* at the *last* $d_i \text{ bits}$, where $d_i = d_0 + i$.
  - $h_{i+1}$ *doubles* the *range* of $h_i$ (*similar to directory doubling*)
Bookkeeping

- Two variables: \textit{Next}, and \textit{Level}
  - \textit{N} points to the \textit{bucket} to be \textit{split next}
  - \textit{L} keeps track of the number of \textit{times} the \textit{range} of the \textit{hash function} has \textit{doubled}

- \textit{Splitting} proceeds in ‘\textit{rounds}’
  - \textit{Round ends} when all \textit{M}_R \textit{initial} (for \textit{round R}) buckets are \textit{split}
  - Buckets \textit{0} to \textit{N} – \textit{1} have \textit{been split}
  - Buckets \textit{N} to \textit{M}_R have yet \textit{to be split}

- \textit{Current round} is \textit{L}
Storage and indexing

One-dimensional indexing

Search and insert

Search

To *find* bucket for *key K*, find \( h_L(K) \)

- If \( h_L(K) \in [N, \ldots M_R] \), *r belongs here*
- Else, *r could belong* to bucket \( h_L(K) \) or bucket \( h_L(r) + M_R \); we *must apply* \( h_{L+1}(K) \) to find out.

Insert

- *Find bucket* as above, by applying \( h_L \) or \( h_{L+1} \)
- If *bucket* to insert is *full*
  - Add *overflow page* and *insert* entry
  - *(Maybe) Split* bucket \( N \) and increment \( N \)
Linear hashing file

Bucket to be split Next

Buckets that existed at beginning of round; range of $h_L$

Buckets split in this round; if $h_L$ falls here, must use $h_{L+1}$

Split image buckets created through splitting of other buckets in this round
Splitting a bucket (0 in this case)

\[ N \]

0 \rightarrow 120 \rightarrow 345 \rightarrow 605 \rightarrow 770

1 \rightarrow 311 \rightarrow 606

2 \rightarrow \ldots

3 \rightarrow \ldots

4 \rightarrow \ldots

Hash functions

- \[ h_0(K) = K \mod 5 \]
Splitting a bucket (0 in this case)

Hash functions

\[ h_1(K) = K \mod 10 \]
Algorithms in more detail

**Lookup for key** $K$

\[
\text{bucket} := h_L(K);
\]

if $\text{bucket} < N$ then $\text{bucket} = h_{L+1}(K)$

**Expansion**

\[
N := N + 1;
\]

if $N = M^{2^L}$ then

\[
L := L + 1; \quad N := 0;
\]

**Contraction**

\[
N := N - 1;
\]

if $N < 0$ then

\[
L := L - 1; \quad N := M^{2^L} - 1;
\]
The expansion process (round 0)

Expansion

\[ N := N + 1; \]

if \( N = M2^L \) then

\[ L := L + 1; \quad N := 0; \]
Linear hashing observations

- Can choose *any criterion* to *trigger split*
  - *Typically*, we want to maintain some *load factor*
- Since *buckets* are *split round-robin*, *long* overflow *chains do not develop!*
- *Doubling* of *directory* in *extendible hashing* is *similar*
  - *Switching* of *hash functions* is *implicit* in how the *number of bits* examined is *increased*
Why more than one dimensions?

- **Single-dimensional** indexes are not enough
  - Consider a *composite search* key e.g., an index on \( \langle \text{sal}, \text{years} \rangle \)
  - The 2-dimensional space is *linearised*
  - We *sort* entries *first* by *sal* and *then* by *years*

- A **multidimensional index** clusters entries
  - *Exploits nearness* in *multidimensional* space.
  - *Balanced* index structures in *multiple dimensions* are *challenging*
The R-tree

- The \textit{R-tree} is a tree-structured index that remains \textit{balanced} on insertions and deletions.
- Each \textit{key} stored in a \textit{leaf entry} is intuitively a \textit{box}, or collection of \textit{intervals}, with \textit{one} interval \textit{per dimension}.

Root of the R-tree

Leaf level
R-tree properties

- **Leaf entry** format: \( \langle n\text{-dimensional bounding box, pointer to record} \rangle \)
  - Bounding box is the tightest bounding box for a data object
- **Non-leaf entry** format: \( \langle n\text{-dim box, pointer to child node} \rangle \)
  - The box covers all boxes in child node (in fact, subtree)
- **All leaves** at same distance from root
- **Nodes** can be kept 50% full (except root)
  - Can choose some parameter \( m \) that is \( \leq 50\% \), and ensure that every node is at least \( m\% \) full
R-tree example
R-tree example (cont.)
Search for objects overlapping box $Q$

Start at \textit{root}

If \textit{current node} is \textit{non-leaf}

\textit{For each} entry $\langle E, \text{ptr} \rangle$, if box $E$ overlaps $Q$, \textit{search subtree} identified by $\text{ptr}$

If \textit{current node} is \textit{leaf}

\textit{For each} entry $\langle E, \text{rid} \rangle$, if $E$ overlaps $Q$, $\text{rid}$ identifies an \textit{object} that \textit{might overlap} $Q$

\textbf{Note}

May have to \textit{search several subtrees} at each node! (In \textit{contrast}, a $B+\text{tree}$ equality search goes to \textit{just one leaf}.)
Insert entry \( \langle B, ptr \rangle \)

Start at root and go down to “best-fit” leaf L

Go to child whose box needs least enlargement to cover \( B \); resolve ties by going to smallest area child

If best-fit leaf \( L \) has space, insert entry and stop. Otherwise, split \( L \) into \( L_1 \) and \( L_2 \)

Adjust entry for \( L \) in its parent so that the box now covers (only) \( L_1 \)

Add an entry (in the parent node of \( L \)) for \( L_2 \). (This could cause the parent node to recursively split.)
Splitting a node

- The *entries* in *node* $L$ plus the *newly inserted* entry must be *distributed* between $L_1$ and $L_2$
- The *Goal* is to *reduce likelihood* of *both* $L_1$ and $L_2$ being *searched* on subsequent queries
- *Redistribute* so as to *minimize area* of $L_1$ plus area of $L_2$

---

**Redistribution**

*Exhaustive* algorithm is *too slow*; *quadratic* and *linear heuristics* are used in practice
Comments on R-trees

- **Deletion** consists of searching for the entry to be deleted, removing it, and if the node becomes under-full, deleting the node and then re-inserting the remaining entries.
- Overall, works quite well for 2- and 3-D datasets.
- Several **variants** (notably, $R+$ and $R^*$ trees) have been proposed; widely used.
- Can improve search performance by using a **convex polygon** to approximate query shape (instead of a bounding box) and testing for polygon-box intersection.
Overview

- Sorting is probably the most classic problem in CS
  - Simple idea: impose a total order on a set of values
- It is a classic problem in databases too
  - Remember ISAM? First step is to sort the file
  - In fact, if you’re bulk loading a B+-tree, you’re better off sorting the file first
- Useful as well for duplicate elimination
- Useful for join evaluation (sort-merge algorithm)
- But what if I have a 1GB relation and 1MB of physical memory?
  - Remember, its all about minimising I/O
  - (Or, why your algorithms class didn’t tell you the whole truth)
Two-way external merge sort

- Requires a **maximum of three buffer pages** and **multiple passes over the data**
- **First pass:** read one page, sort it, write it out
- **Subsequent passes:** read two pages, merge them, write out the result
How it works

- Each pass will read and write each page in the file.
- $N$ pages, so the number of passes is $\lceil \log_2 N \rceil + 1$.
- So, the total I/O cost is $2N(\lceil \log_2 N \rceil + 1)$. 

\[ 2, 3 \]
\[ 4, 6 \]
\[ 2, 6 \]
\[ 3, 4 \]
\[ 4, 9 \]
\[ 7, 8 \]
\[ 5, 6 \]
\[ 3, 1 \]
\[ 2 \]
But why only three pages?

- We have an entire buffer pool of more than three pages, can we utilise it?
  - Yes: N-way merge sort

- To sort a file of $N$ pages using $B$ buffer pool pages:
  - First pass: sorted runs of $B$ pages each ($\lceil \frac{N}{B} \rceil$)
  - Subsequent passes: merge $B-1$ runs (why?)
What is the I/O cost?

- **Number of passes:** \(1 + \lfloor \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \rfloor\)
- **I/O cost:** \(2N \cdot \text{ (Number of passes)}\)
- **For example:** 108 pages in the file, 5 buffer pool pages
  - **Pass 0:** \(\left\lfloor \frac{108}{5} \right\rfloor = 22\) sorted runs of 5 pages each
  - **Pass 1:** \(\left\lfloor \frac{22}{4} \right\rfloor = 6\) sorted runs of 20 pages each
  - **Pass 2:** 2 sorted runs, 80 pages and 28 pages
  - **Pass 3:** final merge, done!
## A bit of perspective

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

\[257 \times 4,096 = 1,052,672\]
Are we done?

- **No!** We can actually *do much better* than this
- **Key observation:** we are using *main memory algorithm* (e.g., quicksort) to sort pages in memory
  - But that *doesn't minimise I/O*, does it?
  - Wouldn’t it be nice if we could *generate sorted runs longer than memory*?
  - **Solution**: *heapsort* (a.k.a. *tournament* or *replacement sort*)
How does heapsort work?

- Keep two heaps in memory, one for each run (the current and the next one).
- Sum of memory needed for the two heaps equals the buffer size.
- Keep adding to the current run until we are out of buffer space.
- When buffer is full, swap heaps and iterate.
The algorithm

**Initialisation**: read $B$ pages into the current heap

while (not finished) do {
    while ($r =$ lowest key from current heap) {
        write $r$ to the current run
        max = $r$
        get $k$ from input
        if ($k > max$) insert $k$ into current heap
        else insert $k$ into next heap
    }
    swap current and next heaps, max = 0
}
Heapsort observations

- What is the average length of a run?
  - Proven to be $2B$ (!)
- Quicksort is computationally cheaper
- But heapsort produces longer runs
  - Minimises I/O
  - Remember, you should “forget” main memory methods when it comes to databases!
Good-old B⁺-trees

- What if the *table to be sorted* has a *B⁺-tree index on sort field*?
  - *Traverse* the *leaf pages* and *we’re done*!
    - Follow the *left-most pointers*, find the *low key*, *scan forward*
  - *Is this always a good idea?*
    - If the *B⁺-tree is clustered*, it’s a *great idea*
    - *Otherwise*, it could lead to *random I/O*
Clustered vs. unclustered storage

Clustered means one sequential scan (good)

Unclustered means random I/O (bad)
Summary of sorting

- *Databases* spend *a lot of their time sorting*
- In fact, they might *dedicate part of their buffer pool* for sorting data
  - Remember *pinning buffer pool pages*?
- *External sorting* minimises *I/O* cost
  - *First* you produce *sorted runs*, *then* you *merge* them
- The *choice of internal sort matters* as well
  - Yes, *quicksort* is *computationally cheap*
  - Though *heapsort* is *computationally more expensive*, it *produces longer runs*, which means *less I/O*
- Finally, *clustered B+trees* (when they exist) are a good way of *sorting in one sequential scan*
Overview

- A physical plan is what the query engine uses in order to evaluate queries.
- In most cases, it is a tree of physical operators:
  - Physical in the sense that they access and manipulate the raw, physical data.
- Plenty of ways to formulate this tree:
  - Identifying the “best” tree is the job of the query optimiser.
Query cycle

Query evaluation

Overview

Query cycle

Catalog

Parse

Query Analyzer

Statistics

Optimiser

Scheduler

Execution environment

DB Tables

Results

Compiled code

Thread-id=12, operator=3, in-stream=1,...

Relational algebra

\[ \sigma_{R.a=5} \]

\[ \pi_{T.b} \]

\[ \land_{R.a=T.b} \]

Query

Syntax tree

Query

Parser

Query

Query Engine

DB Tables

Results

Stratis D. Viglas (University of Edinburgh)

Advanced Databases

86 / 1
Algebraic operators vs. physical operators

- A relational algebraic operator is a procedural abstraction of what should be retrieved.
- The physical operator specifies how the retrieval will take place.
- The same algebraic operator may map to multiple physical operators.
- Physical operators for the same algebraic operator may be implemented using different algorithms.
  - For instance: join → physical join → sort-merge join
Example

SQL query

```sql
select student.id, student.name
from student, course
where student.cid = course.cid and
  course.name = 'ADBS'
```

Algebraic expression

\[
\pi_{\text{student.id}, \text{student.name}}
(\text{student} \bowtie_{\text{student.cid} = \text{course.cid}}
  \sigma_{\text{course.name} = 'ADBS'} (\text{course}))
\]
Mappings to/of various physical operators

- π \(\text{projection list}\)
- σ \(\text{predicate}\)
- \(\pi\) \(\text{(projection list)}\)
- σ \(\text{(predicate)}\)

- tuple-level \(\text{predicate}\)
- block-level \(\text{predicate}\)
- index-level \(\text{predicate}\)

- nested-loops \(\text{predicate}\)

- sort-merge \(\text{predicate}\)

- hash-join \(\text{predicate}\)

- grace-hash \(\text{predicate}\)
- hybrid-hash \(\text{predicate}\)
- symmetric-hash \(\text{predicate}\)

- file-scan \(\text{table}\)
- index-scan \(\text{table}\)
- index-scan \(\text{table-attribute}\)
Math analogy

- Remember *factoring*?
- Same *arithmetic expression* can be *evaluated in different ways*
- If you map arithmetic expressions to *infix notation*, you have *different “plans”*

\[(A+C+FGB+CB+FGA) = C(A+B) + FG(A+B) = (A+B)(C+FG)\]
Physical plans

- **Physical plans** are *trees of physical operators over the physical data*
  - Just as *arithmetic expressions* are *trees of arithmetic operators* over *numbers*
- There are **different ways** of *organising trees of physical operators*
  - Just as there are **different ways** to *organise a mathematical expression*
- **Physical plans** are what *produce query results*
Here’s a plan

**SQL query**

```sql
select student.id, student.name
from student, course
where student.cid = course.cid and course.name = 'ADBS'
```

![Query execution plan diagram]

- **Project**
  - `project student.id, student.name`
- **Select**
  - `select course.name='ADBS'`
  - `select course.cid=student.cid`
- **Join**
  - `course × student`
- **File-scan**
  - `file-scan (course)`
  - `file-scan (student)`
Here’s a better plan

**SQL query**

```sql
select    student.id, student.name
from      student, course
where     student.cid = course.cid and
course.name = 'ADBS'
```
Observations

- Certain selection predicates can be incorporated into the access method.
- If a field is not needed, it is thrown out (why?)
- More than two sources need to be combined (even through a Cartesian product).
- The query plan includes operators not present in the original query.
- Yes, the query specifies what should be retrieved.
  - But how it is retrieved is an entirely different business.
Issues

- *Choice of* order in which *the physical operators are executed*
  - Heuristics, access methods, *optimisation*
- *Choice of algorithms* whenever there are more than one
  - Again, *optimisation* (*join enumeration*, mainly)
- How are *physical operators connected*?
  - Different *execution models*
- What does a *connection* actually *imply*?
  - *Pipelining* (sometimes)
- What about *multiple readers* or even *concurrent updates* of the data?
  - *Concurrency control* (be patient . . .)
- Finally, *how is it all executed*?
  - *Query engine*
A note on duplicates

- The *relational model* calls for *sets of tuples*
- The *query language* (SQL) *does not*
  - Remember “distinct”?  
- *Sets* can be *guaranteed on base relations* by specifying *key (integrity) constraints*
- But what happens with *intermediate results*?
  - *Set semantics are lost*, intermediate results have *bag semantics*
  - But *set semantics* can always *be imposed*; they are just more *expensive to ensure*
Types of plan

- There are two types of plan, according to their shape
  - Deep (left or right)
  - Bushy

- Different shapes for different objectives
Plan objectives

- A *deep plan* is better for *pipelining*
  - Because, let’s face it, *it’s a line!*
- A *bushy plan* is better for *parallel computation*
  - *Different branches* can be *executed concurrently*
- But all of these *depend* on the *algorithms chosen*
  - And on the *execution model*
Summary

- A *plan* is what the *query engine* accepts as *input*
  - ... and what *produces* the *query results*
- The *same algebraic expression* can produce *multiple plans*
  - Because the *same algebraic operator* maps to *multiple physical operators*
- A *physical operator* implements an *evaluation algorithm*
- A *physical plan* does *not necessarily contain* all the *algebraic operators* of the query
  - More or fewer, depending on the *available physical operators*
- The *optimiser chooses* the *best physical plan*
- *Types* of plans are *classified* according to their *shape* and *evaluation objectives*
Overview

- **Physical plans** are *trees of connected physical operators*
- The *execution model* defines the *interface of the connections*
  - And *how data is propagated* from one operator to the next
- It also defines *how operators are scheduled* by the query engine
  - Different *execution models* map to different *process execution paradigms*
Operator connections

- Operator **functionality**: `relation in`, `relation out`
- The **connections** are the **interface** through which the **input** is **read** and **propagated**
- In fact, there is a **producer/consumer** analogy
Pipelining

- **Pipelining** is the following process: *read, process, propagate*
- The *opposite* is to *materialise intermediate results*
- Pipelining *works in theory*, but *in practice* certain *intermediate relations* need to be *materialised*
  - This is called *blocking* (e.g., sorting)
- The benefits of pipelining include
  - *No buffering*
    - *No intermediate relation* is *materialised*
  - *Faster evaluation*
    - Since nothing is materialised, *no disk I/O*
  - *Better resource utilisation*
    - No disk I/O means more *in-memory operation*
What happens in practice

- **Pipelining** is *simulated* through the *operator interface*
- But *different operations* have *different evaluation times*
  - So there will be *some need for buffering*
- If we have *joins*, chances are the *plan will block*
  - We will see *why* that happens when talking about *join algorithms*
The iterator model

- *Also* known as a *cursor*
- *Three basic calls*
  - `open()`
  - `get_next()`
  - `close()`
- Have you ever accessed a database through external code?
  - For example: `exec sql declare cursor in embedded SQL, ResultSet in Java/JDBC, etc.`
Call propagation

- All calls are propagated downstream
- The query engine makes the calls to the topmost operator only
Pure implementation

- The *iterator interface*, as described, is a *completely synchronous* interface.
- A *pure implementation* means that all *operators reside in the same process space*.
  - So *calls* can be *propagated downstream*.
- But *certain operators* are “*faster*” than others.
  - It *could be* the case that an *asynchronous implementation* could be *more beneficial*. 
Different implementations

- The *iterator interface* is what *operators* use to *communicate*
- But *how it is implemented*, can be *entirely different*
  - The *reason* is that there might be *need for buffering*
  - *Three possibilities*
    - *Push model* (buffering in the *operator making the calls*)
    - *Pull model* (buffering in the *operator accepting the calls*)
    - *Streams* (buffering in the *connections*)
The push model

- Tuple *propagation begins* at the *lower levels* of the evaluation tree.
- A *lower operator propagates* a tuple *as soon as it is done* with it.
  - *Does not “care”* if the *receiving operator* has called *get_next()*.
Buffering

The main issue: what happens if the lower operator has propagated the tuple before the operator above it has called get_next()?

Incoming tuples are buffered in the calling operator.

Secondary issue: what should the size of the buffer be? (the optimiser might have an idea...)
The pull model

- The *inverse* of the push model
- If the *lower operator* is *done processing* a *tuple* it *does not propagate* it
  - It *waits* until the *operator above it makes* a *get_next()* call
Buffering — again

outgoing tuples are buffered in the operator being called

operator
tuple

The question this time: what happens if the lower operator is done processing the tuple before the operator above it calls get_next()?

same question: what should the size of the buffer be? (again, the optimiser might have an idea...)
The stream model

- The *connections* become *first-class citizens*
- *Streams* are *queues of tuples* connecting the operators
- *Propagations* and *get_next()* calls are *synchronised* on each stream
Buffering — third time

- *This time*, there is *no question*!
- When the *lower operator* is *done*, it *propagates* the tuple
- When the *top operator* is *ready*, it calls `get_next()` on the incoming stream
Why all this?

- **Pure iterator** implementation
  - If an operator receives `get_next()` and is not ready, it blocks
  - In fact, the entire plan blocks (why?)
  - Assume there is a sort operation somewhere in the plan
    - Congratulations, your plan is officially blocked

- **Non-pure** implementations
  - Operators act (almost) independently of one another
  - Depending on the implementation of the interface (push-, pull-, stream-based) there are different benefits
    - There could still be blocking, but the time during which a plan is blocked is minimised
  - It could lead to each operator running in its own process thread
    - Though this is not always a good idea
Benefits of each model

- **Push model**
  - *Minimises idle time* of the operators (why?)
  - *Great* for *pipelining*

- **Pull model**
  - *Closest* to a *pure implementation*
  - But *still on-demand*

- **Streams model**
  - Fully *asynchronous* to the operators, the *synchronisation* is *on* the *streams*
  - Highly *parallelisable*
Summary

- A **physical plan** is a tree of **connected operators**
- **Operators** need to **communicate data** to one another
- The **iterator interface** is the **means** of this **communication**
  - open(), close(), get_next()
- As with any **interface** there are **different ways of implementing** it, known as **execution models**
  - Push model
    - *Data propagated as soon as* they are **available**
  - Pull model
    - *Data retrieved on demand*
  - Stream model
    - *Asynchronous communication* on the **connections** between operators
Overview

- The join operation is everywhere
  - Any single query with two or more sources will need to have a join (even in the form of a Cartesian product)
  - So common that certain DBMSs implement join indexes
- As a consequence, a DBMS spends a lot of time evaluating joins
- Probably the most optimised physical operator
- A physical operator can be mapped to different algorithms
- As is always the case, a good join algorithm minimises I/O
- Choosing a join algorithm is not as straightforward; the choice might depend on
  - The cardinality of the input, its properties (clustered, sorted, etc.) and any available indexes
  - Available memory
Overview (cont.)

- **Choosing** how to evaluate a single join is different than choosing the order in which joins should be evaluated.

- The *query optimiser* spends most of its time enumerating (ordering) the joins in a query.
  - In fact, the order in which joins are evaluated affects the choice of algorithm.
  - The two are largely interconnected (more on that when discussing query optimisation).
Three classes of algorithms

- **Iteration-based**
  - Namely, *nested loops join* (in three flavours)

- **Order-based**
  - *Sort-merge join* (essentially, merging two sorted relations)

- **Partition-based**
  - *Hash join* (again, in three flavours)
Terminology

- We want to evaluate $R \bowtie S$, shorthand for $R.a = S.b$
  - Also known as an equi-join
- In algebra: $R \bowtie S = S \bowtie R$
  - Not true for the physical join: $\text{cost}(R \bowtie S) \neq \text{cost}(S \bowtie R)$
- Three factors to take into account
  - Input cardinality in tuples $T_R$ and pages $P_R$
  - Selectivity factor of the predicate
    - Think of it as the percentage of the Cartesian product propagated
  - Available memory
Nested loops join

- The simplest way to evaluate a join
- But it can still be optimised so that it minimises I/O
- Very useful for non-equi joins (the other two approaches will not work)
- Three variations
  - Tuple-level nested loops
  - Block-level nested loops
  - Index nested loops
It doesn’t get simpler than this...

**Tuple-level nested loops**

\[
\text{for each tuple } r \in R \text{ do}
\]
\[
\quad \text{for each tuple } s \in S \text{ do}
\]
\[
\quad \quad \text{if } r.a == s.b \text{ then add } \langle r, s \rangle \text{ to the result}
\]

- \( R \) is the *outer relation*
- \( S \) is the *inner relation*
What is the cost?

- **One scan** over the *outer relation*
- **For every tuple** in the *outer relation*, **one scan** over the *inner relation*
- If relations are *not clustered*, then
  - $\text{cost}(R \bowtie S) = T_R + T_R \cdot T_S$
    - Assume $T_R = 100,000$, $T_S = 50,000$, then $\text{cost} = 5,000,100,000 \text{ I/Os}$
    - At 10ms an I/O, that is **50,001,000 seconds**, or, **14,000 hours**
What about clustered storage?

- *Much, much better*, I/O is at a *page level*
- So, the *total cost* will be
  - \( \text{cost}(R \bowtie S) = P_R + P_R \cdot P_S \)
  - In the previous example, for 100 tuples per page, then \( P_R = 1,000, \ P_S = 500, \ \text{cost} = 501,000 \text{ I/Os} \)
  - At 10ms an I/O, that is *5010 seconds*, or *about an hour and a half*
- But we can *improve* that *even more!*
  - *Block-level I/O* and the *buffer pool* will *work wonders*
Here’s an idea

- Assume we have \( B \) pages available in the buffer pool

- *Read as many outer relation pages as possible*; this constitutes a **block**
  - Put the pages of the block in the buffer pool, **pin** them

- *Read the inner relation in pages*

- **Block size** is \( B - 2 \) pages (why?)

- Even more *I/O savings*
The Algorithm

Block-level nested loops

Assumption: \( B \) dedicated pages in the buffer pool, block size is \( B - 2 \) pages

\[
\begin{align*}
\text{for each block of } B - 2 \text{ pages of } R \text{ do} \\
\quad \text{for each page of } S \text{ do } \\
\quad \quad \text{for all matching in-memory tuples } r \in R\text{-block and } s \in S\text{-page} \\
\quad \quad \quad \text{add } \langle r, s \rangle \text{ to result}
\end{align*}
\]
How it works

Disk

main memory (holds the buffer pool)

Result

read a block from $R$ ($B-2$ pages)

read a page from $S$

build a hash table for the block

lookup

matches

output page

Stratis D. Viglas (University of Edinburgh)
How much does it cost?

- The outer relation is **still scanned once** ($P_R$ pages)
- The inner relation is **scanned** $\left\lceil \frac{P_R}{B-2} \right\rceil$ times
  - Each scan costs $P_S$ I/Os
  - So, $\text{cost}(R \bowtie S) = P_R + P_S \cdot \left\lceil \frac{P_R}{B-2} \right\rceil$
  - Same example, $P_R = 1,000$, $P_S = 500$, assume a block size of 100 pages, then **number of I/Os is 6,500**
  - At 10ms per I/O, it will take **65 seconds**
Key observation

- The *inner relation* is *scanned* a number of *times* that is *dependent on* the *size* of the *outer relation*
- So, the *outer relation* should be the *smaller one*
- Let’s *forget the ceilings* and assume two relations: $\text{big}$ and $\text{small}$
- Then we are *comparing*
  - $\text{big} + \text{small} \cdot \frac{\text{big}}{B-2}$
  - $\text{small} + \text{big} \cdot \frac{\text{small}}{B-2}$
- And $\text{big} > \text{small}$
- *Remember*, $\text{cost}(R \bowtie S) \neq \text{cost}(S \bowtie R)$ when it comes to *physical operators*
What if there is an index?

- Suppose the *inner relation* has an *index on the join attribute*
- We can *use the index* to *evaluate the join*
  - Remember, the *join predicate*, if we fix one of the join attribute values, is *just a selection*
- *Scan the outer relation*
  - Look at the *join attribute’s value* and use it to *perform an index lookup* on the *inner relation*
The algorithm

Index nested loops

Assumption: there is an index on $S.b$

\[
\text{for each tuple } r \in R \text{ do} \\
\quad \text{for each tuple } s \in S \text{ where } r.a == s.b \\
\quad \text{add } \langle r, s \rangle \text{ to the result}
\]

- Predicate evaluation is an index lookup in the index over $S.b$
What is the cost?

- Depending on whether the outer relation is **clustered or not**, $P_R$ or $T_R$ I/Os to scan it.
- Selectivity factor $f$: percentage of the **Cartesian product propagated**; this means that every outer tuple joins with $f \cdot T_S$ tuples.
  - Depending on the index, each *lookup* will be, say, $\text{avg\_lookup}$ I/Os.
- If $R$ is **clustered**
  - $\text{cost}(R \bowtie S) = P_R + T_R \cdot f \cdot T_S \cdot \text{avg\_lookup}$
- If $R$ is **not clustered**
  - $\text{cost}(R \bowtie S) = T_R + T_R \cdot f \cdot T_S \cdot \text{avg\_lookup}$
Index nested loops

- If the *selectivity factor* and the *average lookup cost* are *small*, then the *cost* is *essentially a (few) scan(s)* of the *outer relation*
- If the *outer relation* is the *smaller one*, it leads to *significant I/O savings*
- Again, it is the *job* of the *query optimiser* to *figure out if this is the case*
Sort-merge join

- Really *simple idea*
- The *join* is *evaluated* in *two phases*
  - *First*, the two *input relations* are *sorted* on the *join attribute*
  - *Then*, they are *merged* and join *results* are *propagated*
- *External sorting* can be used to *sort* the *input relations*
- The *merging phase* is a *straightforward generalisation* of the *merging phase* used in *merge-sort*
How it works

- Key idea: there exist groups in the sorted relations with the same value for the join attribute.
- We need to take that into account when merging.
  - The reason is that we will have to do some backtracking when generating the complete result.
The algorithm

Merge-join

\[ r \in R, s \in S, gs \in S \]

while (more tuples in inputs) do {
  while (\( r.a < gs.b \)) do advance \( r \)
  while (\( r.a > gs.b \)) do advance \( gs \)  // a group might begin here
  while (\( r.a == gs.b \)) do {
    \( s = gs \)  // mark group beginning
    while (\( r.a == s.b \)) do  // while in group
      add \( \langle r, s \rangle \) to the result; advance \( s \)  // produce result
      advance \( r \)  // move forward
  }
  \( gs = s \)  // candidate to begin next group
}

Stratis D. Viglas (University of Edinburgh) Advanced Databases
What is the cost?

- We know the *cost* of *externally sorting* either *relation*: $2 \cdot P_R \cdot \log P_R$, or $2 \cdot P_S \cdot \log P_S$
- The *merge phase* is essentially *one scan* of *each sorted input*: $P_R$ or $P_S$ (these scans are always clustered)
- $\text{cost}(R \bowtie S) = P_R \cdot (2 \cdot \log P_R + 1) + P_S \cdot (2 \cdot \log P_S + 1)$
  - Running example: $P_R = 1,000$, $P_S = 500$, 100 buffer pool pages to sort, the *number of I/Os is 7,500*
  - At 10ms an I/O, this is *one minute and fifteen seconds* (about the same as block nested loops)
A few issues

- If there are *large groups* in the *two relations*, then we *may* have to *do a lot of backtracking*
  - *Performance will suffer* due to *possible extra I/O*
  - *Hopefully, pages* will be in the *buffer pool*
- *Most relations* can be *sorted* in *2-3 passes*
  - Which *means* that we can *compute the join* in *4 passes max* *(almost regardless of input size!)*
  - In fact, we can *combine* the *final merge of external sorting* with the *merging phase* of the *join* and save even *more I/Os*
Hash join

- *Partition-based* join algorithms
- Key idea: *partition* $R$ and $S$ into $m$ partitions, $R_i$ and $S_i$, so that every $R_i$ fits in memory
  - Observation: *joining tuples* will fall into the *same partition*
- Then, *for every* $R_i$ *load it in memory, scan* $S_i$ *and produce the join results*
- Three flavours: *Simple* hash join, *grace* hash join, *hybrid* hash join
The simple algorithm

Simple hash join

**Assumption:** \( m \) partitions, each partition \( P_i \) fits in main memory

for all partitions \( P_i, i \in [1, m] \)

for each \( r \in R \) read \( r \) and apply hash function \( h_1(r.a) \)

if \( r \) falls into \( P_i \) apply hash function \( h_2(r.a) \) and put it in an in-memory hash table for \( P_i \)

otherwise, write it back out to disk

for each \( s \in S \) read \( s \) and apply hash function \( h_1(s.b) \)

if \( s \) falls into \( P_i \) apply hash function \( h_2(s.b) \) and for all matching tuples \( r \in P_i \), add \( \langle r, s \rangle \) to the result

otherwise, write it back out to disk
How it works — partitioning $R$, iteration $i$

1. **Disk**: Read $R$ and use $h_1(R.a)$
2. **Hash table for $R_i$**: Use $h_2(R.a)$
3. **Output page**: If it falls in $R_i$, write to buffer for all other partitions
4. **Buffer**: When buffer full, write to disk
5. **Output page**: Write to disk

**Main memory (holds the buffer pool)**

**Result**
How it works — partitioning and joining $S$, iteration $i$

main memory (holds the buffer pool)

read $S$
use $h_1(S.b)$

Disk

Hash table for $R_i$

use $h_2(S.b)$
if it falls in $S_i$
writes to buffer for all other partitions

matches

write to disk

matches

Result

buffer

output page

when buffer full
write to disk
What is the cost?

- **Assume equal partition** sizes, input $T$, $P_T$ pages
- For *m partitions*, we will make *m passes* over each input
  - For the first pass:
    - Read $P_T$ pages, write $P_T - \frac{P_T}{m}$ pages: $2P_T - \frac{P_T}{m}$ I/Os
  - For the second pass:
    - Read $P_T - \frac{P_T}{m}$, write $P_T - \frac{P_T}{m} + P_T - 2\frac{P_T}{m}$ pages: $2P_T - 3\frac{P_T}{m}$ I/Os
  - Pass $i$: $2P_T - (2i - 1) \frac{P_T}{m}$ I/Os
- In the end, $m(m + 1)P_T$ I/Os
- For two relations $R$ and $S$, *total cost* is $m(m + 1)(P_R + P_S)$
- Makes sense if *m is small*, or we have *a lot of memory*
- Effectively, this is nested loops join
  - *But* the number of *iterations* is decided by the *number of partitions, not the input sizes!* 

Stratis D. Viglas  (University of Edinburgh)
The “grace” algorithm

Grace hash join

\[
\begin{align*}
\text{for each } r \in R &\text{ read } r \text{ and add it to the buffer page for } h_1(r.a) \\
\text{for each } s \in S &\text{ read } s \text{ and add it to the buffer page for } h_1(s.b) \\
\text{for } i = 1, \ldots, m \text{ do } &\{} \\
\text{for each } r \in R_i &\text{ read } r \text{ and insert it into a hash table using } h_2(r.a) \\
\text{for each } s \in S_i &\text{ do } \{} \\
\text{read } s, \text{ probe the hash table using } h_2(s.b) &\text{ for all matching tuples } r \in R_i \text{ add } \langle r, s \rangle \text{ to the result} \\
\text{clear hash table} &\} \\
\end{align*}
\]
How it works — partitioning \( R \)

read \( R \)
use \( h_1(R.a) \)

main memory (holds the buffer pool)

\( R_1 \) \( R_2 \) \( \ldots \) \( R_m \)

as soon as a page for \( R_i \) fills up
write it to disk
How it works — partitioning $S$

main memory (holds the buffer pool)

read $S$
use $h_1(S.b)$

Disk

$S_1$ $S_2$ ... $S_m$

as soon as a page for $S_i$ fills up
write it to disk
How it works — joining

- Read $R_i$ from disk
  - Use $h_2(R.a)$
- Read $S_i$ from disk
  - Use $h_2(S.b)$
- Hash table for $R_i$
  - Lookup
  - Matches
  - Output page
- Main memory (holds the buffer pool)
  - Result
What is the cost?

- *Scan R* and *write* it to disk, so $2 \cdot P_R$
- Do the *same for S*, so $2 \cdot P_S$
- *Read R* in *partition-by-partition*, so $P_R$
- *Scan S* *partition-by-partition* and *probe* for matches, so $P_S$
- $cost(R \Join S) = 3 \cdot (P_R + P_S)$
  - Same example, $P_R = 1,000$, $P_S = 500$, cost is **4,500 I/Os**
  - At 10ms an I/O the join will take **45 seconds** to evaluate
Memory requirements

- **Objective:** the hash table for a partition must fit in memory
  - Minimise partition size by maximising number of partitions

- What are the optimum sizes?
  - For $B$ buffer pool pages, maximum number of partitions $m = B - 1$ (why?)

- **Size of each partition** is $\lceil \frac{P_R}{B-1} \rceil$

- **Size of the hash table** is $\lceil \frac{f \cdot P_R}{B-1} \rceil$ ($f = \text{fudge factor}$ to capture the increase in partition size due to the hash table)

- During the *probing phase*, in addition to the hash table, we need *one page to read S*, plus *one page for output*
  - So, $B > \lceil \frac{f \cdot P_R}{B-1} \rceil + 2 \Rightarrow B > \sqrt{f \cdot P_R}$
Hybrid hash join

- An *improvement over hash join* if there is *extra memory*
  - Minimum amount of memory for hash join $B > \sqrt{f \cdot P_R}$
  - Suppose that $B > \frac{f \cdot P_R}{k}$, for some integer $k$
  - Divide $R$ into $k$ partitions of size $\frac{P_R}{k}$ (*$k + 1$ buffer pool pages needed*)
  - This leaves $B - (k + 1)$ *extra buffer pool pages*
How it works

- **Suppose** that \( B - (k + 1) > \frac{f \cdot P_R}{k} \)
  - We have *enough memory* during partitioning to *hold* an *in-memory hash table* of size \( B - (k + 1) \) pages
- Idea: *keep \( R_1 \) in memory at all times*
- *While partitioning \( S \), if a tuple falls into \( S_1 \), don’t write* it to disk; instead *probe* the *hash table for \( R_1 \)* for matches
- For all *partitions \( R_i, S_i, i > 2 \), continue as in hash join*
How it works — partitioning and joining

- Read $S$ and use $h_1(S.b)$
- Use $h_2(S.b)$ if tuple falls in $S_1$

Main memory (holds the buffer pool)

- Hash table for $R_1$
- $R_2$, $R_m$

Output page

As soon as page for $S_i$ ($i>1$) fills up, flush to disk

Result

Disk

$R_2$, $R_m$

Matches

Stratis D. Viglas (University of Edinburgh)
Savings over grace hash join

- Essentially, *reduces* the *number* of *full passes*
- Running example, $P_R = 1,000$, $P_S = 500$, assume 300 pages in the buffer pool
- *Choose* the *smaller relation*, $S$
- *Two partitions* for it, *each 250 pages*
  - But *one* will *stay in memory*; so, cost is $500 + 250 = 750$ I/Os
- *Scan* $R$, use *two partitions*, *each 500 pages*
  - But the *first one* is *not written* to disk; so cost is $1,000 + 500 = 1500$ I/Os
- *Join* the *two on-disk partitions*, cost $250 + 500 = 750$ I/Os
- Total cost $750 + 1500 + 750 = 300$ I/Os
- At 10ms an I/O, this is *half a minute*
On predicates

- The algorithms we talked about will work on equi-join predicates
  - If there are no equi-join predicates (inequality joins) the only algorithm that will work is nested loops (why?)
  - If there are indexes on the inequality join predicate’s attributes, we can use index nested loops and revert the join to multiple scans
    - Hoping that we will have buffer pool hits
    - Remember access patterns and page replacement policy?
  - Luckily, in a typical query workload there will mostly be equi-join predicates
On pipelining

- **Pipelining** is great, but it cannot always be achieved.
- All three algorithms will essentially block at some point:
  - In the best case, between matches
  - In the worst case, until after a few scans of the input relations
- This is not necessarily bad; in fact, even if the algorithms block, the time needed to compute the complete join result might be less.
- In reality, more than two stages of pipelining can rarely be obtained in a single plan.
Summary

- The **physical join** is the *most optimised physical evaluation operator*
  - Because a *DBMS spends most of its time evaluating joins*
- **Three main classes** of algorithms
  - *Iteration-based, order-based, partition-based*
- **Three main choice criteria**
  - *Physical layout, indexes, available memory*
Summary (cont.)

- **Iteration-based** methods
  - Essentially, *nested loops*
  - Very *simple to implement*, but if *implemented poorly* very *inefficient*
  - But also *very useful* because they *evaluate non-equi-join predicates*

- **Order-based** methods
  - *Sort* the inputs, *merge* them afterwards
  - *Well-behaved cost* — 3-4 passes over the data will do the trick
Summary (cont.)

- **Partition-based** methods
  - Simple hash join, Grace hash join, and hybrid hash join
  - If there is extra memory, hybrid hash join’s behaviour is excellent

- **Figuring out** the best join algorithm for a particular pair of inputs is the job of the query optimiser

- Which, along with good implementations, will choose the one that evaluates a join in 30 seconds and not in 14,000 hours
Query cycle
Query optimiser

- The *query optimiser* is the *heart* of the *evaluation engine*
  - Yes, the *physical operators* get the *job done*
  - Yes, the *execution model* makes sure the *operators* actually *run*
  - But, unless the *query optimiser decides on those things*, the query will never run
  - And the *decision* needs to be a *good one*
Decisions

- **Two crucial decisions** the optimiser makes
  - The *order* in which the *physical operators* are applied on the inputs (*i.e.*, the plan employed)
  - The *algorithms* that *implement* the *physical operators*

- **These two decisions are not independent**
  - In fact, *one affects the other* in *more ways than one*
Cost-based query optimisation

- The paradigm employed is **cost-based query optimisation**
  - Simply put: enumerate alternative plans, estimate the cost of each plan, pick the plan with the minimum cost
- For **cost-based optimisation**, we need a **cost model**
  - Since what “hurts” performance is I/O, the cost model should use I/O as its basis
  - Hence, the **cardinality-based cost model**
    - **Cardinality** is the number of tuples in a relation
Plan enumeration

- Plan enumeration consists of two parts (again, not necessarily independent from one another)
  - Access method selection (i.e., what is the best way to access a relation that appears in the query?)
  - Join enumeration (i.e., what is the best algorithm to join two relations, and when should we apply it?)

- Access methods, join algorithms and their various combinations define a search space
  - The search space can be huge
  - Plan enumeration is the exploration of this search space
Search space exploration

- As was stated, the *search space* is *huge*
  - *Exhaustive exploration* is *out of the question*
  - Because it *could be the case* that *exploring* the search space might *take longer than* actually *evaluating* the query
  - The *way* in which we *explore* the *search space* describes a *query optimisation method*
    - *Dynamic programming*, *rule-based* optimisation, *randomised* exploration, . . .
Just an idea...

- A query over *five relations*, only *one access method*, only *one join algorithm*, only *left-deep plans*
  - Remember, $\text{cost}(R \bowtie S) \neq \text{cost}(S \bowtie R)$
  - So, the number of *possible plans* is $5! = 120$
  - If we add *one extra access method*, the number of *possible plans* becomes $2^5 \cdot 5! = 3840$
  - If we add one *extra join algorithm*, the number of *possible plans* becomes $2^4 \cdot 2^5 \cdot 5! = 61440$
Cardinality-based cost model

A *cardinality-based cost model* means we need *good ways of* doing the following

- *Using cardinalities* to *estimate costs* (e.g., accurate cost functions)
- *Estimating output cardinalities after* we apply *certain operations* (e.g., after a selection the cardinality will change; it will not change after a projection)
  - *Because* these *output cardinalities will be used as inputs* to the *cost functions* of other operations
Cardinality estimation

- An entire area of query optimisation
- Largely a matter of statistics
- It has triggered the “percentage wars”
  - “This estimation technique is within $x\%$ of the true value with a $y\%$ probability”
- **Fact**: the better the statistics, the better the decisions
- **Another fact**: errors in statistics propagate exponentially; after 4 or 5 joins, you might as well flip a coin
- **Third fact**: cost functions are discontinuous, so in certain scenarios only perfect statistics will help
Are we done?

- The *previous issues* were only a *subset* of the *problems* an *optimiser* solves
  - We also need to *worry* about *certain properties* of the data
    ✪ For instance, if we *use a B+tree* as an access method, then *we won’t have to sort* (e.g., interesting orders in System R)
    ✪ If we use a *hash join later on* the *order is spoiled*
    ✪ So we will *have to sort* again
  - *Depending* on the *algorithm* and the *environment*, we need to *allocate memory*
- And as if *all these were not enough*, *optimisation time assumptions* do *not necessarily hold* at *run time*
The final nail . . .

- These are all for one query
- Now, imagine a system doing that for 1000 queries
  - Simultaneously
- And it all has to be done fast
  - Once a decision is made, it cannot be undone
Conclusion

- **Query optimisation** is a *very, very hard problem*
- But **without it** a **DBMS** is **doomed** to **seriously sub-optimal performance**
- The **problem** is **not nearly solved**
  - All we **have** is **decent optimisation strategies**
  - And **decent sub-problem solutions**
- **Fact:** **rarely** will an **optimiser** **pick** the “**best**” **plan**
  - But it will **almost always pick** a **plan** with **good performance** and **stay away** from **bad choices**
  - At the end of the day, that’s what counts
The agenda

- **Mapping SQL queries to relational algebra**
  - Query blocks, uncorrelated vs. correlated queries
- **Optimisation of a single query block**
- **Equivalence rules**
- **Statistics and cardinality estimation**
- **Search space exploration**
  - Dynamic programming (System-R)
SQL decomposition

- **SQL queries** are optimised by *decomposing* them into a *collection* of *query blocks*
- A *block* is optimised in *isolation*, resulting in a *plan* for a *block*
- *Plans* for *blocks* are combined to form the *complete plan* for the query
What is a block?

- An *SQL query* with *no nesting*
- Exactly *one select-clause*
- Exactly *one from-clause*
- *At most one*
  - *Where-clause* in *conjunctive normal form*
  - *Group by-/sort by-*clause
  - *Having-*clause
Example

Sample schema

- **Sailors** (sid, sname, rating, age)
- **Boats** (bid, bname, color)
- **Reserves** (sid, bid, day, rname)

Example

For each sailor with the highest rating over all sailors, and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

SQL query

```sql
select s.sid, min(r.day)
from sailors s, reserves r, boats b
where s.sid = r.sid and r.bid = b.bid and
  b.color = 'red' and
  s.rating = (select max(s2.rating) from sailors s2)
group by s.sid
having count(*) > 1
```
Two blocks in the query

```
select s.sid, min(r.day)
from sailors s, reserves r, boats b
where s.sid = r.sid and r.bid = b.bid and
       b.color = 'red' and
       s.rating = ( select max(s2.rating)
                     from   sailors s2 )
group by s.sid
having count(*) > 1
```

```
select max(s2.rating)
from   sailors s2
```
Single block optimisation — step 1

SQL query

```sql
select s.sid, min(r.day)
from sailors s, reserves r, boats b
where s.sid = r.sid and r.bid = b.bid and
  b.color = 'red' and
  s.rating = ( select max(s2.rating) from sailors s2)
group by s.sid
having count(*) > 1
```

Relational algebra

\[
\pi_{s.sid, \min(r.day)}(
    \text{having} \ count(*) > 2(
    \text{group by} \ s.sid(
        \sigma_{s.sid = r.sid \land r.bid = b.bid \land b.color = \text{red} \land s.rating = \text{nested-value}(
            sailors \times reserves \times boats))))
)\]
Single block optimisation — step 2

Relational algebra — before

$$\pi_{s.sid, \min(r.day)}(\sigma_{s.sid=r.sid \land r.bid=b.bid \land b.color=red \land s.rating=\text{nested\text{-}value}}(\pi_{s.sid, \min(r.day)}(\sigma_{s.sid=r.sid \land r.bid=b.bid \land b.color=red \land s.rating=\text{nested\text{-}value}}(sailors \times reserves \times boats))))$$

Relational algebra — after

$$\pi_{s.sid}(\sigma_{s.sid=r.sid \land r.bid=b.bid \land b.color=red \land s.rating=\text{nested\text{-}value}}(sailors \times reserves \times boats)))$$

- **Ignore the aggregate operations**
  - They **only** have **meaning** for the complete result
  - **Convert** the query into a **subset** of **relational algebra** called \( \sigma \pi \times \)
Single block optimisation — step 3

- Use *equivalence rules* to identify *alternative ways* of *formulating the query*
- “*Plug in*” *algorithms*
- *Enumerate plans*
- *Estimate* the *cost* of each *plan*
- *Pick* the *one* with the *minimum cost*
Equivalence rules

- Essentially, every query block consists of three things:
  - Cartesian product of all relations in the from-clause
  - Selection predicates of the where-clause
  - Projections of the select-clause

- The equivalence rules define the space of alternative plans considered by an optimiser.
  - In other words, the search space of a query
Selection and projections

- **Cascading of selections**
  \[ \sigma_{c_1 \land c_2 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R)))) \]

- **Commutativity**
  \[ \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \]

- **Cascading of projections**
  \[ \pi_{a_1}(R) \equiv \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R))\ldots)) \]
  \[ \text{iff } a_i \subseteq a_{i+1}, \ i = 1, 2, \ldots n - 1 \]
Cartesian products and joins

- **Commutativity**
  - $R \times S \equiv S \times R$
  - $R \bowtie S \equiv S \bowtie R$

- **Assosiativity**
  - $R \times (S \times T) \equiv (R \times S) \times T$
  - $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$

- **Their combination**
  - $R \bowtie (S \bowtie T) \equiv R \bowtie (T \bowtie S) \equiv (R \bowtie T) \bowtie S$
  - $\equiv (T \bowtie R) \bowtie S$
Among operations

- **Selection-projection commutativity**
  - $\pi_a ( \sigma_c(R)) \equiv \sigma_c ( \pi_a(R))$
  - iff *every attribute in* $c$ *is included in the set of attributes* $a$

- **Combination (join definition)**
  - $\sigma_c (R \times S) \equiv R \bowtie c S$

- **Selection-Cartesian/join commutativity**
  - $\sigma_c (R \times S) \equiv \sigma_c(R) \bowtie S$
  - iff the *attributes in* $c$ *appear only in* $R$ *and not in* $S$

- **Selection distribution/replacement**
  - $\sigma_c(R \bowtie S) \equiv \sigma_{c_1 \land c_2} (R \bowtie S) \equiv \sigma_{c_1} (\sigma_{c_2}(R \bowtie S)) \equiv \sigma_{c_1}(R) \bowtie \sigma_{c_2}(S)$
  - iff $c_1$ *is relevant only to* $R$ *and* $c_2$ *is relevant only to* $S$
Among operations (cont.)

- **Projection-Cartesian product commutativity**
  - $\pi_a( R \times S ) \equiv \pi_{a_1}(R) \times \pi_{a_2}(R)$
  - iff $a_1$ is the subset of attributes in $a$ appearing in $R$ and $a_2$ is the subset of attributes in $a$ appearing in $S$

- **Projection-join commutativity**
  - $\pi_a( R \bowtie_c S ) \equiv \pi_{a_1}(R) \bowtie_c \pi_{a_2}(R)$
  - iff same as before and every attribute in $c$ appears in $a$

- **Attribute elimination**
  - $\pi_a( R \bowtie_c S ) \equiv \pi_{a}( \pi_{a_1}(R) \bowtie_c \pi_{a_2}(S) )$
  - iff $a_1$ subset of attributes in $R$ appearing in either $a$ or $c$ and $a_2$ is the subset of attributes in $S$ appearing in either $a$ or $c$
What do we have and what do we need?

- **We have**
  - A way to *decompose SQL queries* into *multiple query blocks*
  - A way to *map a block* to *relational algebra*
  - *Equivalence rules* between different *algebraic expressions*, i.e., a search space

- **We need**
  - A way to *estimate the cost* of *each alternative expression*
    - *Depending* on the *algorithms* used
  - A way to *explore* the *search space*
Cost estimation

- A **plan** is a tree of operators
- **Two parts** to estimating the **cost** of a plan
  - For each node, estimate the **cost** of performing the corresponding **operation**
  - For each node, estimate the **size** of the **result** and any **properties** it might have (e.g., sorted)
- **Combine** the **estimates** and **produce** an **estimate** for the entire **plan**
Cost and cardinality

- We have seen *various storage methods* and *algorithms*
  - And *know the cost* of *using each* one, *depending* on the *input cardinality*
- The *problem is estimating* the *output cardinality* of the *operations*
  - Namely, *selections* and *joins*
Selectivity factor

- The **maximum number of tuples** in the **result** of any **query** is the **product** of the **cardinalities** of the **participating relations**

- Every **predicate** in the **where-clause** **eliminates some** of these **potential results**

- **Selectivity factor** of a **single predicate** is the **ratio** of the **expected result size** to the **maximum result size**

- **Total result size** is **estimated** as the **maximum size times** the **product** of the **selectivity factors**

- **Key assumption**: the **predicates** are statistically **independent**
How it works

SQL query

```
select a_1, a_2, ..., a_k
from R_1, R_2, ..., R_n
where P_1 and P_2 and ... and P_m
```

Maximum output cardinality

```
|R_1| · |R_2| · · · · |R_n|
```

Selectivity factor product

```
f_{P_1} · f_{P_2} · · · · f_{P_m}
```

Estimated output cardinality

```
(f_{P_1} · f_{P_2} · · · · f_{P_m}) · |R_1| · |R_2| · · · · |R_n|
```
Various selectivity factors

- **column = value** → \(\frac{1}{\#\text{keys}(column)}\)
  - Assumes *uniform distribution* in the values
  - Is itself an *approximation*

- **column_1 = column_2** → \(\frac{1}{\max(\#\text{keys}(column_1), \#\text{keys}(column_2))}\)
  - *Each value* in *column_1* has a *matching value* in *column_2*; *given* a *value* in *column_1*, the *predicate* is just a *selection*
  - Again, an *approximation*
Various selectivity factors (cont.)

- \( \text{column} > \text{value} \rightarrow \frac{(\text{high(column)} - \text{value})}{(\text{high(column)} - \text{low(column)})} \)
- \( \text{value}_1 < \text{column} < \text{value}_2 \rightarrow \frac{(\text{value}_2 - \text{value}_1)}{(\text{high(column)} - \text{low(column)})} \)
- \( \text{column in list} \rightarrow \text{number of items in list times s.f. of column} = \text{value} \)
- \( \text{column in sub-query} \rightarrow \text{ratio of subquery’s estimated size to the number of keys in column} \)
- \( \text{not (predicate)} \rightarrow 1 - (\text{s.f. of predicate}) \)
- \( P_1 \lor P_2 \rightarrow f_{P_1} + f_{P_2} - f_{P_1} \cdot f_{P_2} \)
Key assumptions made

- The *values across columns* are *uncorrelated*
- The *values* in a *single column* follow a *uniform distribution*
- *Both* of these assumptions *rarely hold*
- The *first assumption* is *hard to lift*
  - Only recently have researchers started tackling the problem
- The *uniform distribution* assumption can be *lifted* with *better statistical methods*
  - In our case, *histograms*
What we would like

Estimated distribution (uniform assumption) vs True distribution

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.25</td>
<td>4.50</td>
<td>6.75</td>
<td>9.00</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
Lifting the uniform distribution assumption

- At the *basic level*, all we *need* is a *collection* of *(value, frequency)* pairs
- Which is *just a relation*!
  - So, *scan* the *input* and *build* it
- But this is *unacceptable*
  - Because the *size* might be *comparable* to the *size* of the *relation*
  - And we *need* to *answer* *queries* about the *value distribution* fast

<table>
<thead>
<tr>
<th>name</th>
<th>color</th>
<th>stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>bolt</td>
<td>red</td>
<td>10</td>
</tr>
<tr>
<td>bolt</td>
<td>green</td>
<td>5</td>
</tr>
<tr>
<td>nut</td>
<td>blue</td>
<td>4</td>
</tr>
<tr>
<td>nut</td>
<td>black</td>
<td>10</td>
</tr>
<tr>
<td>nut</td>
<td>red</td>
<td>5</td>
</tr>
<tr>
<td>nut</td>
<td>green</td>
<td>10</td>
</tr>
<tr>
<td>cam</td>
<td>blue</td>
<td>5</td>
</tr>
<tr>
<td>cam</td>
<td>green</td>
<td>10</td>
</tr>
<tr>
<td>cam</td>
<td>black</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>3</td>
</tr>
<tr>
<td>blue</td>
<td>2</td>
</tr>
<tr>
<td>black</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Histograms

- **Elegant data structures** to *capture value distributions*
  - *Not affected* by the *uniform distribution* assumption (though this is *not entirely true*)
- They offer *trade-offs* between *size* and *accuracy*
  - The *more memory* that is dedicated to a histogram, the *more accurate* it is
  - But also, the *more expensive* to manipulate
- *Two* basic classes: *equi-width* and *equi-depth*
Desirable histogram properties

- **Small**
  - Typically, a DBMS will allocate a *single page* for a histogram!

- **Accurate**
  - Typically, less than *5% error*

- **Fast access**
  - *Single lookup* access and *simple algorithms*
Mathematical properties

- A *histogram approximates* the *value distribution* for *attribute X* of *table T*
- The *value distribution* is *partitioned* into a number of *b subsets*, called *buckets*
- There is a *partitioning constraint* that *identifies how* the *partitioning* takes place
  - Different constraints, lead to *different classes* of histograms
- The *values* and *frequencies* in *each bucket* are *approximated* in some *common fashion*
Equi-width histogram

True distribution

Distribution approximation

Bucket 1: count = 8
Bucket 2: count = 4
Bucket 3: count = 15
Bucket 4: count = 3
Bucket 5: count = 15
Equi-width histogram construction

- The **total range** is **divided** into **sub-ranges** of **equal width**
- Each **sub-range** becomes a **bucket**
- The **total number of tuples** in **each bucket** is **stored**

<table>
<thead>
<tr>
<th>min</th>
<th>max</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Equi-width histogram estimation

- To estimate the output cardinality of a range query
  - The starting bucket is identified
  - The histogram is then scanned forward until the ending bucket is identified
  - The numbers of tuples in the buckets of the range are summed
  - Within each bucket the uniform distribution assumption is made

- $6 \leq v \leq 10$: $\frac{3}{3} \cdot 15 + \frac{2}{3} \cdot 3 = 17$
Equi-depth histogram

- True distribution
- Distribution approximation

Bucket 1: count = 8
Bucket 2: count = 10
Bucket 3: count = 10
Bucket 4: count = 7
Bucket 5: count = 9
Equi-depth histogram construction and estimation

- The total range is divided into sub-ranges so that the number of tuples in each range is (approximately) equal.
- Each sub-range becomes a bucket.
- The same schema as in equi-width histograms is used.
- In fact, the same algorithm is used for estimation (!).
- $6 \leq v \leq 10$: $\frac{2}{4} \cdot 10 + \frac{2}{2} \cdot 10 + \frac{1}{4} \cdot 7 \approx 17$
Comparison

- **Equi-depth** histograms are *generally better* than *equi-width*
  - **Buckets** with *frequently occurring values* contain *fewer values*
  - *Infrequently occurring values* are approximated *less accurately (but the error is less significant)*
  - So the *uniform distribution assumption within* each *bucket* leads to *better approximation*
What do we have and what do we need?

- **We have**
  - A way to *decompose* a *query*
  - A way to *identify* equivalent, *alternative representations* of it (*i.e.*, a *search space*)
  - A *statistical framework* to *estimate* cardinalities
  - A *cost model* to *estimate* the *cost* of an alternative

- **We need**
  - A way to *explore* the *search space*
  - *Dynamic programming*
Dynamic programming

- In the beginning, there was *System R*, which had an *optimiser*
- *System R’s optimiser* was using *dynamic programming*
  - An *efficient way* of exploring the search space
- *Heuristics*: use the *equivalence rules* to *push down selections* and *projections*, *delay Cartesian products*
  - *Minimise input cardinality* to, and *memory* requirements of the *joins*
- *Constraints*: *left-deep plans*, *nested loops* and *sort-merge join* only
  - *Left-deep plans* took better *advantage of pipelining*
  - *Hash-join* had *not* been *developed* back then
Interesting orders

- If there is an *order by* or *group by* clause on an *attribute*, we say that this *attribute* has an *interesting order* associated with it.
  - *Interesting*, because depending on the *access method* we can get away with *fewer physical operations* (e.g., sorting).
- The *same holds* for *attributes* participating in a *join*.
  - Again, *interesting* because we can *use* the *access method* in *evaluating* the join.
Dynamic programming steps

- Identify the cheapest way to access every single relation in the query, applying local predicates
  - For every relation, keep the cheapest access method overall and the cheapest access method for an interesting order
- For every access method, and for every join predicate, find the cheapest way to join in a second relation
  - For every join result keep the cheapest plan overall and the cheapest plan in an interesting order
- Join in the rest of the relations using the same principle
An example

<table>
<thead>
<tr>
<th>name</th>
<th>dno</th>
<th>job</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>50</td>
<td>12</td>
<td>8500</td>
</tr>
<tr>
<td>Jones</td>
<td>50</td>
<td>5</td>
<td>15000</td>
</tr>
<tr>
<td>Doe</td>
<td>51</td>
<td>5</td>
<td>9500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dno</th>
<th>dname</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>MFG</td>
<td>Edinburgh</td>
</tr>
<tr>
<td>51</td>
<td>Billing</td>
<td>London</td>
</tr>
<tr>
<td>52</td>
<td>Shipping</td>
<td>Glasgow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>job</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>clerk</td>
</tr>
<tr>
<td>6</td>
<td>typist</td>
</tr>
<tr>
<td>8</td>
<td>sales</td>
</tr>
<tr>
<td>12</td>
<td>mechanic</td>
</tr>
</tbody>
</table>

The query is:

```sql
select name, title, salary, dname
from emp, dept, job
where job.title = 'Clerk' and 
     dept.location = 'Edinburgh' and 
     emp.dno = dept.dno and 
     emp.job = job.job
```

**local predicates**

**join predicates**

**interesting orders**

---

Stratis D. Viglas (University of Edinburgh)
Access methods and local predicates

Scanning emp is the most expensive method for emp; emp.dno and emp.job are interesting orders.

Scanning dept is the most expensive method for dept; dept.dno is an interesting order.

Scanning job is the cheapest method for job; but, job.job is an interesting order.
Search tree for access methods
Join enumeration for relation \textit{emp} (nested loops join)

- Both \textit{emp} \Join \textit{dept} results are \textit{in different interesting orders} so they are propagated.
- Only the \textit{cheapest result} in any \textit{interesting order} is propagated for \textit{each pair of inputs}.
Join enumeration for relations dept, job (nested loops)

- \(\text{cost}(\text{emp} \bowtie \text{dept}) \neq \text{cost}(\text{dept} \bowtie \text{emp})\) so we will enumerate dept’s joins even though we have an alternative for generating the same result (same for job \(\bowtie\) emp).
- Both dept \(\bowtie\) emp results in the same order, only one propagated.
- Since there is no dept \(\bowtie\) job predicate in the query, that join is not enumerated (same for job \(\bowtie\) dept).
- The unordered result for job \(\bowtie\) emp is propagated because it is the cheapest overall.
Search tree — 2 relations, nested loops join

Stratis D. Viglas (University of Edinburgh)
Join enumeration for relation **emp** (sort-merge)
Join enumeration for relations *dept, job* (sort-merge)
Search tree — 2 relations, sort-merge join

- **emp** × **dept**
  - Index: (emp.dno)
  - Merge: (dept.dno)
  - Cost: \( \text{cost}(\text{emp.dno}) + \text{cost}(\text{dept.dno}) + \text{cost-m}(\text{emp} \bowtie \text{dept}) \)
  - Order: dno order

- **emp** × **job**
  - Index: (emp.job)
  - Cost: \( \text{cost}(\text{emp.job}) + \text{cost}(\text{job.job}) + \text{cost-m}(\text{emp} \bowtie \text{job}) \)
  - Order: job order

- **dept** × **emp**
  - Index: (dept.dno)
  - Merge: (emp.dno)
  - Cost: \( \text{cost}(\text{dept.dno}) + \text{cost}(\text{emp.dno}) + \text{cost-m}(\text{dept} \bowtie \text{emp}) \)
  - Order: dno order

- **job** × **emp**
  - Index: (job.job)
  - Cost: \( \text{cost}(\text{job.job}) + \text{cost}(\text{emp.job}) + \text{cost-m}(\text{job} \bowtie \text{emp}) \)
  - Order: job order
Search tree — 2 relations, both join methods

For each pair of relations, for each different join order and for each interesting order for that pair one plan is propagated.

An unordered result is only propagated if it is the cheapest overall for a pair in a given join order.
Three relations

- **Repeat** the process
  - For *every pair* of two relations
  - For *every join* method
  - For *every access method* of the *remaining relation*
  - *Find* the *cheapest way* to *join* the *third relation* with the *pair*
    - *Estimate* cardinalities
    - *Estimate* the *cost* of computing the *join*
  - *Keep* the *cheapest choice* for *every interesting order* and the *cheapest* for the *unordered* case *if* it is the *cheapest overall*
Rule-based optimisation

- **Basically** an issue of *if-then rules*
  - *If* *(condition list)* *then apply some transformation* to the plan constructed so far
    - *Estimate* the *cost* of the *new plan, keep it only if* it is *cheaper than* the original
  - The *order* in which the *rules are applied* is *significant*
  - As a *consequence*, rules are applied *by precedence*
    - *For instance, pushing down selections* is given *high precedence*
    - *Combining two relations with a Cartesian product* is given *low precedence*
Randomised exploration

- **Mostly useful** in *big queries* (more than 15 joins or so)
- The *problem* is one of *exploring a bigger portion* of the search space
  - So, *every once in a while* the optimiser “jumps” to some *other part* of the search space *with some probability*
- As a *consequence*, it gets to *explore parts* of the search space it would *not have explored otherwise*
The “well”

cost

comparable, but good performance

diverse, but bad performance

"plans"
The “well” and local minima

cost

diverse, but bad performance

local minima

comparable, but good performance

"plans"
Final step — the entire plan

- The optimizer has produced plans for each query block.
- The question is now one of combining the sub-plans to formulate the entire query plan.
- The strategy used depends on whether the outer and nested queries are correlated or not.
  - If they are, then in all probability the two sub-plans will be combined through a join.
Uncorrelated queries

- **Usually**, they can be **executed in isolation**
- The **nested query feeds** the **outer query with results**

```sql
select s.sname
from   sailors s
where  s.rating = (select max(s2.rating)
                   from sailors s2)
```

executed first

once the nested query is executed,
the outer is simply a selection
Correlated queries

- Sometimes, it is **not possible** to **execute** the **nested query just once**
- In those cases the **optimiser reverts** to a **nested loops** approach
  - The **nested query** is **executed** once for **every tuple** of the **outer query**

```sql
select s.sname
from sailors s
where exists (select *
              from reserves r
              where r.bid = 103 and s.sid = r.sid)
⋈
s.sid=r.sid
σr.bid=103
reserves
sailors
nested loops
this could be
an entire plan
```
In practice

- **Before** breaking up the *query into blocks*, most systems try to *rewrite* the *query* in some *other way* (*de-correlation*)
  - The idea is that *there will probably be a join*, so it will be *better* if the *query* is *optimised in its entirety*
- If *de-correlation* is *not possible*, then it is *nested loops all the way*
  - Usually, *compute* the *nested query*, *store* it in a temporary relation and *do nested loops* with the *outer*
What do we have and what do we need?

- **We have**
  - A way to *decompose* a *query*
  - A way to *identify* equivalent, *alternative representations* of it (*i.e.*, a *search space*)
  - A *statistical framework* to *estimate* cardinalities
  - A *cost model* to *estimate* the *cost* of an alternative
  - Ways of *exploring* the *search space*

- **We need**
  - *Nothing!*
Summary

- The **query optimiser** is the **heart** of the **query engine**
  - If it does not do a **good job**, the engine is doomed to **sub-optimal performance**
- **Two** key, closely related **decisions**
  - **Order** in which **operations** are performed
  - **Algorithms** that perform the **operations**
- The **paradigm** used is **cost-based optimisation**
  - **Three steps**: generate alternative plans, estimate the cost of each plan, pick the cheapest
- The **cost model** used is the **cardinality-based** cost model
  - Because **cardinality** is a **good I/O metric**
  - As a **consequence**, we need **good ways of doing** two things
    - Estimating the **cost** of an **algorithm**
    - Estimating the **output cardinality** of **operations**
Summary (cont.)

- **Cardinality estimation** is 50% of the problem
  - Two approaches: **uniform distribution assumption**, or **histograms**
  - The **uniform distribution** assumption essentially does *not* “care” about the values themselves, they all have an **equal probability of appearing**
  - **Histograms** are a **better** and **more elegant** distribution **approximation technique**
    - **Equi-width** and **equi-depth** histograms are the two dominant classes
The remaining 50% is search space exploration

- Largely based on the equivalence rules of relational algebra
- Dynamic programming is the dominant approach
  - Find the cheapest way to access single relations
  - Find the cheapest way to join two relations
  - For each pair, find the cheapest way to join in a third relation
  - Keep going . . .
Other approaches include rule-based optimisation, randomised exploration, ...  

All approaches aim at one thing  
- Picking a good evaluation plan  
- It might not be the cheapest overall, but it usually is of comparable cost  

Query optimisation is still an open issue  
- We have good ways of solving sub-problems, but the entire problem remains largely unsolved
Overview

- So far, we have *focused* on *query processing*
  - In other words, *reading* and *manipulating* data
- A *database system*, however, *not only reads, but also stores* data
  - *At the same time* as others are *querying* it
- We *need* a way to *ensure concurrent access* to the data
  - *Without compromising* system *performance*
Overview (cont.)

- The **basic concept** is *transaction processing*
- *Every transaction* needs to satisfy **four basic properties**
  - Atomicity, *consistency*, *isolation*, *durability*
- *How* does the system *guarantee* these *properties*?
  - Remember, *without compromising performance*
  - *Solution*: by *interleaving transactions*
Overview (cont.)

- How can we decide if, after we have interleaved transactions, the result is correct?
  - Interleaving transactions actually causes certain anomalies
  - Solution: the system uses locks to ensure correctness

- How are locks used?
  - Lock granularity, degrees of consistency and two-phase locking

- What impact do locks have on performance?
Overview (cont.)

- **Locking** poses significant *overhead*
  - Luckily, however, this *overhead* can be "tuned" by the user
  - *Transaction isolation* levels

- But what if the *worse comes to worst*?
  - *System crashes*
  - *Transactional semantics* and *recovery*
  - *Write-ahead logging* and the *ARIES algorithms*
Transactions

- A DBMS spends a lot of time waiting on I/O
  - It is important to keep the CPU busy while waiting
  - In other words, execute other operations concurrently
- Fact: the DBMS does not “care” what the user does with the data that is being read or written
  - All it cares about is that data is being read or written
- A transaction is the DBMS’s abstract view of user programs: a sequence of reads and writes
Concurrent execution

- The transaction user abstraction: when a user submits a transaction it is as if the transaction is executing by itself
  - The DBMS achieves concurrency by interleaving transactions
  - If the transaction begins with the DB in a consistent state, it must leave the DB in a consistent state after it finishes
- The semantics of the transactions are unknown to the system
  - Whether the transaction updates a bank account or it fires a rocket missile, the DBMS will never know!
ACID properties

- **Atomicity**: all the actions in a transaction are executed as a single atomic operation; either they are all carried out or none are.

- **Consistency**: if a transaction begins with the DB in a consistent state, it must finish with the DB in a consistent state.

- **Isolation**: a transaction should execute as if it is the only one executing; it is protected (isolated) from the effects of concurrently running transactions.

- **Durability**: if a transaction has been successfully completed, its effects should be permanent.
Example

- Consider *two transactions*
  - *First* transaction *transfers* funds, *second* transaction *pays* 6% *interest*

- *If* they are *submitted* at the *same time*, there is *no guarantee* as to *which* is *executed first*
  - But the *end effect* should be *equivalent* to the *transactions running serially*
### Example (cont.)

#### Acceptable schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>A = A+100</th>
<th>B = B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>A = 1.06*A</td>
<td>B = 1.06*B</td>
</tr>
</tbody>
</table>

#### Problematic schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>A = A+100</th>
<th>B = B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>A = 1.06*A</td>
<td>B = 1.06*B</td>
</tr>
</tbody>
</table>

#### DBMS's view

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A), W(A)</th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>R(A), W(A)</td>
<td>R(B), W(B)</td>
</tr>
</tbody>
</table>
Scheduling

- A *schedule* is a *sequence* of *reads* and *writes* for some *transaction workload* incorporating all actions of the *workload’s transactions*
  - *Serial schedule*: the *actions* of different transactions are *not interleaved*
  - *Equivalent schedules*: for any *database state*, the *effect* of executing the *first schedule* is *identical* to the *effect* of executing the *second schedule*
  - *Serialisable schedule*: a *schedule* that is *equivalent* to a *serial schedule*
## Conflicts

### Reading uncommitted data (WR conflicts, or "dirty reads")

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A), W(A)</th>
<th></th>
<th>R(B), W(B), A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>R(A), W(A)</td>
<td>R(B), W(B), C</td>
<td></td>
</tr>
</tbody>
</table>

### Unrepeatable reads (RW conflicts)

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A)</th>
<th></th>
<th>R(A), W(A), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>R(A)</td>
<td>W(A), C</td>
<td></td>
</tr>
</tbody>
</table>

### Overwriting uncommitted data (WW conflicts, or "lost updates")

<table>
<thead>
<tr>
<th>T1</th>
<th>W(A)</th>
<th></th>
<th>W(B), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>W(A)</td>
<td>W(B), C</td>
<td></td>
</tr>
</tbody>
</table>

---

Stratis D. Viglas (University of Edinburgh)  
Advanced Databases
The solution: locks

- Before a transaction “touches” a DB object it has to obtain a lock for it
  - S (Shared) lock for reading
  - X (eXclusive) lock for writing

- Strict two-phase locking (Strict 2PL)
  - Each transaction must obtain an S lock for everything it reads before it starts reading it and an X lock for everything it writes before it starts writing
  - All locks held by a transaction are released only when the transaction commits
  - Once a transaction obtains an X lock for a DB object no other transaction can obtain an X or an S lock for that object

- Strict 2PL produces only serialisable schedules
What can go wrong?

- If a transaction $T_i$ is aborted, then all its actions have to be undone; not only that, but if $T_j$ reads an object written by $T_i$, $T_j$ needs to be aborted as well (cascading aborts).
- Most systems avoid cascading aborts with the following rule:
  - If $T_i$ writes an object $T_j$ can read, this object can be read only after $T_i$ commits.
- In order to know what needs to be undone, the system keeps a log, recording all writes.
- The log is also helpful when recovering from system crashes.
The log

- The **following actions** are **recorded** in the **log**
  - Whenever a **transaction writes** an **object**
    - The **log record must** be **on disk before** the **data record** reaches the disk
  - Whenever a **transaction commits/aborts**

- **Log records** are **chained** by **transaction ID** (why?)

- All **log-related activities** (in fact, all **concurrency control related activities**) are **handled by the DBMS**
  - The **user does not know anything**
Crash recovery

- **Three phases** to recovery (ARIES)
  - **Analysis**: scan log **forward**, identifying **committed** and **aborted/unfinished** transactions
  - **Redo**: all **committed transactions** are **made durable**
  - **Undo**: the actions of all **aborted** and/or **unfinished transactions** are **undone**
Concurrency control

- **Serial schedule**: the actions of different transactions are not interleaved.
- **Equivalent schedules**: for any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- **Serialisable schedule**: a schedule that is equivalent to a serial schedule.
- Two schedules are conflict equivalent if:
  - They involve the same actions of the same transactions.
  - Every pair of conflicting actions is ordered the same way.
- Schedule $S$ is conflict serialisable if $S$ is conflict equivalent to some serial schedule.
Dependency graphs

Given a schedule $S$

- One node per transaction
- An edge from $T_i$ to $T_j$, if $T_j$ reads or writes an object written by $T_i$

Theorem: a schedule $S$ is conflict serialisable if and only if its dependency graph is acyclic
Example: not conflict serialisable schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A), W(A)</th>
<th></th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>R(A), W(A)</td>
<td>R(B), W(B)</td>
<td></td>
</tr>
</tbody>
</table>

T2 reads A, written by T1
T1 reads B, written by T2
Review: Strict 2PL

- **Strict two-phase locking (Strict 2PL)**
  - Each transaction must obtain an S (Shared) lock for everything it reads before it starts reading it and an X (eXclusive) lock for everything it writes before it starts writing.
  - All locks held by a transaction are released only when the transaction commits.
  - Once a transaction obtains an X lock for a DB object no other transaction can obtain an X or an S lock for that object.

- **Strict 2PL produces only serialisable schedules**
  - In other words: schedules with acyclic dependency graphs.
Simple 2PL

- **Two-phase locking (2PL)**
  - Each transaction must obtain an *S* (Shared) lock for everything it reads before it starts reading it and an *X* (eXclusive) lock for everything it writes before it starts writing.
  - A transaction cannot request additional locks once it releases any locks.
  - Once a transaction obtains an *X* lock for a DB object no other transaction can obtain an *X* or an *S* lock for that object.
Lock management

- *Lock* and *unlock requests* are *handled by the lock manager* that maintains a *lock table*.

- *Lock table entry*:
  - *Number of transactions* currently *holding a lock*
  - *Type of lock* held (*shared* or *exclusive*)
  - *Pointer to queue of lock requests*

- *Locking* and *unlocking* have to be *atomic operations*.

- *Lock upgrade*: *transaction* that *holds* a *shared lock* can be *upgraded* to hold an *exclusive lock*.
Deadlocks

- As *always*, where *there are locks*, *there are deadlocks*
- **Deadlocks**: cycle of transactions waiting for locks to be released by each other
- **Two ways of dealing with deadlocks**
  - Deadlock *prevention*
  - Deadlock *detection*
Deadlock prevention

- The solution involves timestamps; a timestamp is the transaction’s priority.
- If $T_i$ wants a lock that $T_j$ holds, there are two possible policies:
  - **Wait-Die**: if $T_i$ has higher priority, $T_i$ waits for $T_j$; otherwise $T_i$ aborts.
  - **Wound-Wait**: if $T_i$ has higher priority, $T_j$ aborts; otherwise $T_i$ waits.
- If a transaction re-starts, it has its original timestamp.
Deadlock detection

- **Create** a waits-for graph
  - **Nodes** are transactions
  - There is an edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock

- **Periodically check** for cycles in the waits-for graph
Multiple granularity locks

- *What* should we *lock*?
  Tuples, pages, tables, ...

- But there is an *implicit containment*

- *Idea: lock* DB objects *hierarchically*
Hierarchical locks and new locking modes

- **Allow transactions** to lock at each level of the hierarchy
- **Introduce** “intention” locks: IS and IX
  - Before locking an item, a transaction must introduce intention locks on all the item’s ancestors in the hierarchy
  - Release locks in reverse order
- **One extra lock**: SIX — “share, with intention to write”
## Compatibility matrix

<table>
<thead>
<tr>
<th></th>
<th>NL</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>IS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IX</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SIX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>S</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
In more detail

- *Each transaction starts* from the *root* of the *hierarchy*
- To *obtain S or IS lock on a node*, *must hold IS or IX on parent node*
  - What if a transaction holds SIX on parent? S on parent?
- To *obtain X or IX or SIX on a node*, *must hold IX or SIX on parent node*
- *Must release* locks in *bottom-up order*
A few examples

- **T1 scans** $R$, and **updates** a few *tuples*
  - **T1** gets an *SIX lock* on $R$, then *repeatedly* gets an *S lock* on *tuples* of $R$, and *occasionally upgrades* to *X* on the *tuples*

- **T2 uses an index** to **read only** *part of* $R$
  - **T2** gets an *IS lock* on $R$, and *repeatedly* gets an *S lock* on *tuples* of $R$

- **T3 reads all of** $R$
  - **T3** gets an *S lock* on the *entire relation*
  - **Or**, it gets an *IS lock* on $R$, *escalating* to *S lock* on every tuple
Here’s the catch (the phantom problem)

- If we **relax** the **assumption** that the **DB** is a **fixed collection** of objects, **even Strict 2PL** will **not assure serialisability**!
  - **T1 locks all pages** containing **sailor records** with **rating = 1**, and **finds oldest sailor** (say, **age = 71**)
  - Next, **T2 inserts a new sailor**: **rating = 1, age = 96**
  - **T2 also deletes oldest sailor** with **rating = 2** (and, say, age=80), and **commits**
  - **T1 now locks all pages** containing **sailor records** with **rating = 2**, and **finds oldest** (say, age=63)

- **No lock conflicts**, **but also no consistent DB state** where T1 is “correct”!
The problem

- *T1 implicitly assumes* that it has *locked the* set of *all sailor* records with *rating = 1*
  - The *assumption* only *holds if no sailor* records are *added while T1* is *executing*!
  - We *need* some *mechanism* to *enforce* this *assumption*
    - *Index locking*
    - *Predicate locking*

- The *example shows* that *conflict serialisability guarantees serialisability* only *if* the set of *objects* is *fixed*!
Index locking

- **If** there is an *index* on the *rating field*, *T1* should *lock* the *index page* containing the *data entries* with *rating* = 1
  - **If** there are *no records* with *rating* = 1, *T1 must lock* the *index page* where such a *data entry would be*, *if it existed*!

- **If** there is *no suitable index*, *T1 must lock all pages*, and *lock* the *file/table* to *prevent* new *pages* from being *added*, to *ensure* that *no new records* with *rating* = 1 are *added*
Predicate locking

- **Grant lock** on all *records* that *satisfy* some *logical predicate*, e.g., \( \text{salary} > 2 \cdot \text{salary} \)
  - *Index locking* is a *special case* of *predicate locking* for which an *index supports* efficient *implementation* of the *predicate lock*
  - What is the *predicate* in the *sailor example*?

- *In general*, *predicate locking* imposes a *lot of locking overhead*
B⁺-tree locking

- How can we efficiently lock a particular node?
  - This is entirely different than multiple granularity locking (why?)
- One solution: ignore the tree structure, just lock pages while traversing the tree, following 2PL
  - Terrible performance
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root
Key observations

- **Higher levels** of the tree *only direct searches* to leaf pages
- For *insertions*, a *node* on a *path* from the *root* to a modified *leaf* must be *locked* (in *X mode*, of course), *only if* a *split* can *propagate up* to it *from* the *modified leaf* (similar point holds for deletions)
- We can *exploit* these *observations* to design *efficient locking protocols* that *guarantee serialisability* even though they *violate 2PL*
The basic algorithm

- **Search**: start at root and descend; repeatedly, S lock child then unlock parent
- **Insert/Delete**: start at root and descend, obtaining X locks as needed; once child is locked, check if it is safe:
  - Safe node: a node such that changes will not propagate up beyond this node
    - Insertion: node is not full
    - Deletion: node is not half-empty
  - If child is safe, release all locks on ancestors
Example: search 38*

Obtain and release S-locks level-by-level
Example: delete 38*

*Obtain X-locks while descending; release them top-down once the node is designated safe*
Example: insert 25*

Obtain X-locks while descending; leaf-node is not safe so create a new one and lock it in X-mode; first release locks on leaves and then the rest top-down
Optimistic B+-tree locking

- **Search**: as before
- **Insert/delete**: set locks as if for search, get to the leaf, and set X lock on the leaf
  - If the leaf is not safe, release all locks, and restart transaction, using previous insert/delete protocol
- “Gambles” that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful
  - In practice, better than previous algorithm
Example: insert 25*

Obtain S-locks while descending, and X-lock at leaf; the leaf is not safe, so abort, release all locks and restart using the previous algorithm.
Even better algorithm

- **Search**: as before
- **Insert/delete**: use original insert/delete protocol, but set IX locks instead of X locks at all nodes
  - Once leaf is locked, convert all IX locks to X locks top-down: i.e., starting from the unsafe node nearest to root
  - **Top-down reduces** chances of deadlock
    - Remember, this is not the same as multiple granularity locking!
Hybrid approach

- The likelihood that we will need an X lock decreases as we move up the tree.
- Set S locks at high levels, SIX locks at middle levels, X locks at low levels.
# Transaction isolation

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty read</th>
<th>Unrepeatable read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable reads</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serialisable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Review: ACID properties

- **Atomicity**: all the actions in a transaction are executed as a single atomic operation; either they are all carried out or none are.
- **Consistency**: if a transaction begins with the DB in a consistent state, it must finish with the DB in a consistent state.
- **Isolation**: a transaction should execute as if it is the only one executing; it is protected (isolated) from the effects of concurrently running transactions.
- **Durability**: if a transaction has been successfully completed, its effects should be permanent.

Atomicity and durability are ensured by the recovery algorithms.
What can go wrong?

- **Atomicity**
  - Transactions may *abort*; their *effects* need to be *undone*

- **Durability**
  - What if the system *stops running?*

**Transactional semantics**
- $T_1$, $T_2$, $T_3$ should be *durable*
- $T_4$, $T_5$ should be *aborted*
Problem statement

- Updates are happening in place
  - There is a buffer pool
    - Data pages are read from disk
    - Data pages are modified in memory
    - Overwritten on, or deleted from disk
- We need a simple scheme to guarantee atomicity and durability
More on the buffer pool

- Two issues: *force* and *steal*
- **Force**: when a *data page* is *modified* it is *written* straight to *disk*
  - Poor response time
  - But durable
- **Steal**: effects of *uncommitted transactions* reach the *disk*
  - Higher throughput
  - But not atomic
The problems

- **Steal**’s problems are all about *atomicity*
  - What if a *transaction modifying a page aborts*?
  - If we *steal a page*, we need to *remember its old value* so it can be *restored* (*UNDO*)

- **No force**’s problems are all about *durability*
  - What if a *system crashes before a modified page is written to disk*?
  - We need to *record enough information* to make the *changes permanent* (*REDO*)
The solution: logging

- **Record REDO and UNDO information in a record of a separate structure: the log**
  - **Sequential writes** for every update
  - **Minimal information** written (more efficient!)
  - **Keep it on a separate disk!**

- **Log**: a list of REDO and UNDO actions
  - Each log record contains at least:
    - Transaction id, modified page, old data, new data
Write-ahead logging

- The log adheres to the write-ahead protocol (WAL)
  1. Must force the log record for an update before the corresponding data page gets to disk
  2. Must force all log records for a transaction before it commits

- #1 guarantees atomicity
- #2 guarantees durability
Normal execution

- **Series** of *reads* and *writes*
- **Followed** by a *commit* (success) or *abort* (failure)
- **Steal, No-force** management
- **Adherence** to the *WAL protocol*
- **Checkpoints:** *periodically*, the system creates a checkpoint to *minimise* the *time* taken to *recover*
  - Assume the DB is *consistent* *after* a *checkpoint*
WAL and the log

- Each *log record* has a unique *log sequence number* (LSN)
  - LSNs are *always increasing*
- Each *data page* contains a *pageLSN*
  - The LSN of the *most recent log record* for an *update to that page*
- The *system keeps track of flushedLSN*
  - The max LSN flushed so far
- **WAL**: *before a page is written, pageLSN ≤ flushedLSN*
Log records

- Possible log records types
  - **Update**
  - **Commit**
  - **Abort**
  - **End** (signifies commit or abort!)
  - **Compensation Log Records** (CLR)
    - Logging UNDO actions!
    - But we will not talk about them in more detail
Other log-related state

- **Transaction table**: one entry per active transaction
  - Contains transaction id, status (running/committed/aborted) and lastLSN — log sequence number of the last log record for that transaction

- **Dirty page table**: one entry per dirty page in buffer pool
  - Contains recLSN — the LSN of the log record which first caused the page to be dirty
Checkpoint records

- **begin_checkpoint** record: indicates *when checkpoint began*
- **end_checkpoint** record: contains *current transaction table* and *dirty page table*
- This is a "fuzzy checkpoint"
  - Other *transactions continue to run*; so these *tables accurate* only *as of* the *time* of the *begin_checkpoint record*
  - *No attempt* to *force dirty pages* to disk; *effectiveness* of *checkpoint limited* by *oldest unwritten change* to a *dirty page*
  - So it’s a *good idea* to *periodically flush dirty pages* to disk
- Store *LSN of checkpoint record* in a *safe place* (master record)
What’s stored where

- Log records flushed to disk
- Log records
  - prevLSN
  - transID
  - type
  - pageID
  - length
  - offset
  - before-image
  - after-image

- Log record
- Data pages (each with a pageLSN)
- Master record
  - DB
  - main memory
  - transaction table
  - lastLSN
  - status
  - dirty page table
  - recLSN
  - flushedLSN
Simple transaction abort

- For now, consider an *explicit abort* of a transaction
  - *No crash* involved
- We want to “*play back***” the log in *reverse order, UNDO ing updates*
  - Get *lastLSN* of *transaction* from *transaction table*
  - *Follow chain* of *log records backward* via the *prevLSN* field
  - *Before starting UNDO*, write an *Abort log record*
    - For *recovering* from crash *during UNDO!*
Abort (cont.)

- To *perform UNDO*, must have a *lock on data*
  - No problem
- *Before restoring old value* of a page, *write a CLR*
  - *Continue logging* while you *UNDO*!
  - CLR has one *extra field*: undonextLSN
    - Points to the *next LSN to undo* (i.e., the *prevLSN* of the *record we’re currently undoing*)
    - CLR*s are never undone* (but they *might be redone* when repeating history: *guarantees atomicity*)
- At the *end of UNDO*, write an *“end” log record*
Transaction commit

- Write *commit record* to *log*
- All *log records* up to the *transaction’s lastLSN* are *flushed*
  - Guarantees that *flushedLSN ≥ lastLSN*
  - Note that *log flushes* are *sequential, synchronous writes* to disk
  - Many *log records per log page*
- Commit() returns
- Write *end record* to *log*
Recovery: big picture

- **Start** from a *checkpoint* (found via *master record*)
- **Three phases**
  - **Analysis**: figure out which *transactions committed* since the checkpoint, and which *failed*
  - **REDO** all actions
    - *Repeat history*
  - **UNDO** effects of *failed transactions*
Additional issues

- **What happens** if the *system crashes* during the *analysis phase*?
  - During *REDO phase*?
- How can the *amount of work* during *REDO* be *limited*?
  - *Flush asynchronously* in the background
- How can the *amount of work* during *UNDO* be *limited*?
  - *Avoid long-running transactions*
Summary

- **Concurrency control** and **recovery** are **key concepts** of a DBMS
- **Both** are **ensured** by the **system itself**; the user does not (and should not!) know of their existence
- The **key abstraction** is the **transaction**
  - The **processing unit** of the **system**
  - **Four** key **properties**
    - Atomicity, **consistency, isolation, durability**
Summary (cont.)

- A *transaction* is *viewed by the system* as a *series* of *reads* and *writes*.
- To *improve throughput*, the *system interleaves* the *actions* of the *transactions* (i.e., a schedule)
  - At all times, *ensuring serialisability* of the *produced schedules*.
- **Locks** are the *mechanism* that *ensures serialisability*.
  - *Before reading*, obtain a *Shared lock*.
  - *Before writing*, obtain an *eXclusive lock*. 
Summary (cont.)

- **Multiple granularity of locks**
  - Leads to an *escalation of locks*, as we are *descending the hierarchy*

- **Special protocols for indexes and predicates**

- **Transactions** help *after recovering from a crash*
  - As the *processing unit*, we know *what needs to be repeated or deleted*
Summary (cont.)

- **Steal**, no-force buffer pool management
  - *Higher response time* (steal)
  - *Higher throughput* (no-force)

- Need to use it, without satisfying correctness

- Use a log to record all actions
  - Employ the Write-Ahead Logging protocol
Use **checkpoints** to **periodically record consistent states** and **limit** the amount of the **log** that needs to be **scanned during recovery**.

**Recovery in three phases**

- **Analysis**: from checkpoint, **figure out REDO and UNDO extents**
- **REDO**: repeat entire history
- **UNDO**: delete effects of failed transactions

**Repeating history simplifies the logic.**
Why parallelism?

- The very definition of parallelism: divide a big problem into many smaller ones to be solved in parallel
- Consider we have a terabyte of data to scan
  - With one pipe of 10MB/s, we need 1.2 days
  - By partitioning the data in disjoint subsets and having 1,000 parallel pipes of the same bandwidth, we need 90s
Parallelism and DBMSs

- **Parallelism** is natural to DBMS processing
  - **Pipeline** parallelism: *many machines* each doing *one step* in a *multi-step* process
  - **Partition** parallelism: *many machines* doing the *same thing* to *different pieces* of data.
  - *Both* are natural in a DBMS

Partitioning: *split* inputs, *merge* outputs
The parallelism success story

- **DBMSs** are the *most (only?) successful application of parallelism*
  - Teradata, Tandem vs. Thinking Machines, KSR, ...
  - Every *major DBMS vendor* has some *parallel server*
  - *Workstation manufacturers* now depend on *parallel DB server* sales

- **Reasons** for success
  - Bulk-processing (*partition parallelism*)
  - Natural *pipelining*
  - *Inexpensive hardware* can do the trick
  - Users/app-programmers do *not* need to *think in parallel*
**Terminology**

**Speed-up**

*More resources means proportionally less time for given amount of data* (throughput)

**Scale-up**

If *resources increased in proportion to increase in data size*, *time is constant*
Parallel data management

Architecture: what to share?

- **Shared memory**
  - *Easy* to program
  - *Expensive* to build
  - *Difficult* to scale up

- **Shared disk**
  - *Middle* of the road
  - *Distributed* file system
  - *Cluster* computing

- **Shared nothing**
  - *Hard* to program
  - *Cheap* to build
  - *Easy* and *ideal* to speed/scale up
Different types of parallelism

- **Intra-operator** parallelism
  - *All machines* working to compute a *single operation* (scan, sort, join)

- **Inter-operator** parallelism
  - *Each operator* may run *concurrently* on a *different site* (exploits pipelining)

- **Inter-query** parallelism
  - *Different queries* run on *different sites*

- We shall *focus* on *intra-operator* parallelism
Automatic data partitioning

- **Range**: Good for equi-joins, Range-queries, Good for aggregation
- **Hash**: Good for equi-joins, No range-queries, Problematic with skew
- **Round-robin**: Indifferent for equi-joins, Range-queries complicated, Load-balanced
Parallel scans

- *Scan* in *parallel*, and *merge*
- *Selections* may *not require all* sites for *range* or *hash* partitioning
- *Indexes* can be built at *each partition*
- **Question**: how do *indexes differ* in the different *schemes*
  - Think about *both lookups* and *inserts*!
  - What about *key* indexes?
Parallel sorting

- Key idea: sorting *phases* are intrinsically *parallelisable*
  - *Scan* in parallel, *range-partition* as you go
  - As *tuples come in*, begin “local” *sorting* using standard algorithm
  - *Resulting data* is *sorted*, and *range-partitioned*

- Problem: *skew*
  - Solution: *sample* the data to determine *partition points*
Parallel aggregation

- For each aggregate function, need a decomposition
  - \( \text{count}(S) = \sum_i \text{count}(s(i)) \), ditto for \( \text{sum}() \)
  - \( \text{avg}(S) = \frac{(\sum_i \text{sum}(s(i)))}{\sum_i \text{count}(s(i))} \)
  - and so on …

- For groups
  - Sub-aggregate groups close to the source
  - Pass each sub-aggregate to its group’s site
    - Chosen via a hash function
Parallel joins

- **Nested loops**
  - Each outer tuple must be *compared* with each inner tuple that might join
  - Easy for *range partitioning* on join columns, hard otherwise

- **Sort-merge** (or plain *merge-*) join
  - *Sorting* gives *range-partitioning*
  - *Merging* partitioned tables is *local*
Parallel hash join

- During the **first phase**, **partitions** are **distributed** to **different sites**
  - A good **hash function automatically** distributes work **evenly**
- **Second phase** is **local** at each **site**
  - Almost **always** the **winner** for **equi-join**
- **Good use** of **split/merge** makes it **easier** to build **parallel versions** of **sequential join** code
Dataflow network for parallel join
Complex parallel query plans

- **Complex** queries: *inter-operator* parallelism
  - *Pipelining* between operators
    - Note that *sorting* and *phase one* of *hash-join block* the pipeline (yet again!)
  - *Bushy* execution trees
Observations

- It is *relatively easy* to build a *fast parallel query executor*
- It is *hard* to write a robust and world-class *parallel query optimizer*
  - There are many *tricks*
  - One quickly hits the *complexity barrier*
  - Still *open research*
Parallel query optimization

- **Common** approach: *two phases*
  - Pick *best sequential* plan (System R algorithm)
  - Pick *degree of parallelism* based on current system parameters

- **Allocate operators** to processors
  - Take *query tree, decorate* as in previous example
What can go wrong?

- **Best sequential plan ≠ best parallel plan**
- **Trivial counter-example**
  - *Table partitioned* with *local secondary index* at *two nodes*
  - *Range query*: *all of node 1 and 1% of node 2*
    - e.g., *select * from telephone_book where name < "NoGood"
  - *Node 1 should* do a *scan* of its partition
  - *Node 2 should* use secondary *index*

![Diagram showing table scan and index scan](attachment://diagram.png)
Parallel databases summary

- **Parallelism** natural to *query processing*
  - Both *pipeline* and *partition* parallelism
- **Shared-nothing** vs. **Shared-memory**
  - *Shared-disk* too, but *less standard*
  - *Shared-mem* *easy, costly*; does *not scaleup*
  - *Shared-nothing* *cheap, scales* well, *harder* to implement
- **Intra-operator**, **inter-operator**, and **inter-query** parallelism all possible.
Parallel database summary (cont.)

- **Data layout** choices important
- **Most** database **operations** can be done using **partition-parallelism**
  - Sort
  - Sort-merge join, hash-join
- **Complex plans**
  - Allow for **pipeline-parallelism**, but sorts, hashes **block** the **pipeline**
  - **Partition-parallelism** achieved through **bushy trees**
Parallel database summary (cont.)

- **Hardest** part: *optimization*
  - *Two-phase* optimization *simplest*, but can be *ineffective*
  - More *complex schemes* still at the *research* stage

- We have not discussed transactions, logging
  - *Easy* in *shared-memory/shared-disk* architecture
  - Takes *some care* in *shared-nothing*
  - Some ideas from *distributed transactions* are *handy*