Secondary Storage

• Rough Speed Differentials
  – nanoseconds: retrieve data in main memory
  – microseconds: retrieve from disk cache or under a read head
  – milliseconds: retrieve from elsewhere on disk

• Approximate Disk Speeds
  – seek (head move): 8 milliseconds
  – rotational latency (spin under): 4 milliseconds
  – block transfer: 68 microseconds (negligible)
  – total: 12.068 – about 12 milliseconds

• Implications
  – cluster data on cylinders
  – make good use of caches

Sequential Files

• Operations
  – Add: write over a deleted record or after last record
  – Delete: mark deleted
  – Access: read until record found (half the file, on average)

• Sorted (doesn’t help much without an index)
  – Access: binary search (if contiguous)
  – Add: can be expensive to maintain sort order
  – Delete: mark deleted
Indexes

Primary (Key) Index

<table>
<thead>
<tr>
<th>Block#/Offset</th>
<th>Block#</th>
</tr>
</thead>
<tbody>
<tr>
<td>101 1.0</td>
<td>101 1</td>
</tr>
<tr>
<td>102 1.1</td>
<td>123 2</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>123 2.0</td>
<td>763 25</td>
</tr>
</tbody>
</table>

Secondary (Key) Index

<table>
<thead>
<tr>
<th>&lt;Name, Street, City&gt;</th>
<th>Block#</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Smith, 12 Maple, Boston&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;Carter, 10 Main, Hartford&gt;</td>
<td>1</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>&lt;Jones, 20 Main, Boston&gt;</td>
<td>2</td>
</tr>
</tbody>
</table>

Secondary (Nonkey) Index

<table>
<thead>
<tr>
<th>City</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>1, 2, . . . , 25</td>
</tr>
<tr>
<td>Hartford</td>
<td>1, . . . , 25</td>
</tr>
</tbody>
</table>

Dense & Sparse Indexes

Indexed Sequential File

1. Sorted on primary key
2. Sparse index
3. Overflow buckets

Operations:
- Access (ok)
- Delete (ok)
- Insert (ok, if space in block)
Variable-Length Records

<table>
<thead>
<tr>
<th>GuestNr</th>
<th>RoomNr</th>
<th>ArrivalDate</th>
<th>NrDays</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>10 May</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>20 May</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15 May</td>
<td>2</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>10 May</td>
<td>5</td>
</tr>
</tbody>
</table>

Three Implementations:
1. Reserve enough space for maximum.
2. Chain each nested record.
3. Reserve space for the expected number and chain the rest.

Hashing

- Static Hashing
  - similar to in-memory hashing (block/offset addresses)
  - records in contiguous blocks (may be hard to find sufficient)
- Open Hashing
  - hash table of pointers to buckets
  - buckets: chained blocks of dense-index value-pointer pairs
  - operations: retrieve, add, delete

```plaintext
101   Smith   12 Maple   Boston
h(101)
```

Chapter 3 - 5

Chapter 3 - 6
Indexing Verses Hashing

- Store and retrieve on key – hashing wins (usually)
- Search on non-key – indexing wins (usually)
- Range search – indexing wins
- Search on multiple attributes – indexing wins (usually)

However: for highly dynamic updates, indexed-sequential files degenerate quickly – need B*-tree indexes.

B*-Tree Index

Basic Idea

- N-way, balanced tree whose non-root nodes are always at least half full.
- Leaf nodes chained and contain value-pointer pairs whose pointers point at blocks of records.
Chapter 3 - 9

B⁺-Tree Index – Example

Chapter 3 - 10

B⁺-Tree: Insertion & Deletion

- Insertion
  - place in leaf (if space)
  - split and promote to parent (do recursively, if necessary)

- Deletion
  - remove from leaf (if not too few)
  - take from neighbor (if not too few)
  - coalesce and take from parent (do recursively, if necessary)
Query “Optimization”

- Actually: Query Improvement
- Query Rewriting
  - Main Strategy: Make intermediate results small by applying selection and projection early.
  - Additional Strategies: Remove unnecessary operations, execute common subexpressions only once, …
- Cost Estimation & Lowest-Cost Selection
  - Estimate the execution time for two or more execution methods.
  - Select the most efficient.

Rewriting Rules 1 & 2

1. \( \pi_X e = e \)  \quad \text{Note: the scheme of } e \text{ is } X. 

\[ \pi_{\text{RoomNr, Name, NrBeds, Cost}} r = r \]

2. \( \pi_X \sigma_f e = \pi_X \sigma_f \pi_{XY} e \)  \quad \text{Note: } f \text{ mentions the attrs. of } Y. 

\[ \pi_{\text{RoomNr, Name}} \sigma_{\text{Cost > 75}} r = \pi_{\text{RoomNr, Name}} \sigma_{\text{Cost > 75}} \pi_{\text{RoomNr, Name, Cost}} r \]
Rewriting Rules 3 & 4

3. \( \sigma_f(e_1 \times e_2) = \sigma_{f_1}(\sigma_{f_2} e_1 \times \sigma_{f_3} e_2) \)  
   Note: \( f = f_1 \land f_2 \land f_3 \); each pertains to its respective expression.

\[
\sigma_{\text{ArrivalDate}} = 15 \text{ May} \quad \land \quad \sigma_{\text{City}} = \text{Boston} \quad (g \times s)
\]
\[
= \sigma_{\text{City} = \text{Boston} \mid g} \times \sigma_{\text{ArrivalDate} = 15 \text{ May} \mid s}
\]

4. \( \pi_X(e_1 \times e_2) = \pi_X(\pi_{X_1 Y} e_1 \times \pi_{X_2 Y} e_2) \)  
   Note: \( X_1 = X \cap \text{scheme}(e_1) \), \( X_2 = X \cap \text{scheme}(e_2) \), and \( Y = (\text{scheme}(e_1) \cap \text{scheme}(e_2)) - X \).

\[
\pi_{\text{RoomNr}} (r \mid g)
\]
\[
= \pi_{\text{RoomNr}} (\pi_{\text{RoomNr}, \text{Name} \mid r} \times \pi_{\text{Name} \mid g})
\]

Rewriting Rules 5 & 6

5. \( \pi_X \pi_Y e = \pi_X e \)

\[
\pi_{\text{Name}} \pi_{\text{RoomNr}, \text{Name} \mid r} = \pi_{\text{Name} \mid r}
\]

6. \( \sigma_{f_1} \sigma_{f_2} e = \sigma_{f_1 \land f_2} e \)

\[
\sigma_{\text{NrBeds} = 2 \land \text{Cost} = 80} r = \sigma_{\text{NrBeds} = 2} \land \sigma_{\text{Cost} = 80} r
\]
Rewriting Rules 7, 8 & 9

Natural join (\(|\times|\)) is
(7) commutative, (8) associative, and (9) idempotent.

\[(r |\times| (g |\times| s)) |\times| r\]
\[=_{8} r |\times| g |\times| s |\times| r\]
\[=_{7} r |\times| r |\times| g |\times| s\]
\[=_{9} r |\times| g |\times| s\]

Query Rewriting – Example

\[
\Pi_{\text{Name, StreetNr, City}} \sigma_{\text{ArrivalDate }= 10 \text{ May}} (\sigma_{\text{NrBeds }= 2} (r |\times| g |\times| s))
\]
\[=_{3,8} \Pi_{\text{Name, StreetNr, City}} (\sigma_{\text{NrBeds }= 2} r |\times| g |\times| \sigma_{\text{ArrivalDate }= 10 \text{ May}} s)\]
\[=_{4,8} \Pi_{\text{Name, StreetNr, City}} (\Pi_{\text{RoomNr, Name}} (\sigma_{\text{NrBeds }= 2} r |\times| \Pi_{\text{GuestNr, Name, StreetNr, City}} g |\times| \Pi_{\text{GuestNr, RoomNr}} \sigma_{\text{ArrivalDate }= 10 \text{ May}} s))\]
\[=_{1} \Pi_{\text{Name, StreetNr, City}} (\Pi_{\text{RoomNr, Name}} (\sigma_{\text{NrBeds }= 2} r |\times| g |\times| \Pi_{\text{GuestNr, RoomNr}} \sigma_{\text{ArrivalDate }= 10 \text{ May}} s))\]
Cost Estimation

• Based on:
  – operations
  – file and record sizes
  – file structures
  – block layout on disk

• Estimate:
  – number of seeks
  – number of rotational delays
  – number of blocks transferred (ignore unless > 100 or so)
  – (ignore memory-access time and CPU time)

Cost Estimation – Selection Example

\( \sigma_{\text{ArrivalDate}} = 15\text{ May} \)

Assume: 10,000 records, 50/block

1. sequential file, not contiguous, not sorted
   \( \frac{10,000}{50} = 200 \) block accesses
   @ 12 ms per access = 2400 ms = 2.4 sec

2. sequential file, contiguous, not sorted
   With 132 blocks per track, 200 blocks fit on a cylinder;
   1 access + 200 block transfers = 12 ms + 200 \( \times \) 68 \( \mu \)s
   = 12 + 13.6 ms = 25.6 ms

3. sequential file, contiguous, sorted (on ArrivalDate)
   Binary search \( \lceil \log_2 200 \rceil = 8 \); so 1 seek + 8 rotational delays = 8 ms + 8 \( \times \) 4 ms = 40 ms (worse than unsorted)
Cost Estimation – Selection Example

Assume: 10,000 records, 50/block

σ_{ArrivalDate} = 15 May

4. sequential file, not contiguous, sorted (on ArrivalDate), indexed, with index in memory

   If considerably fewer than 50, all records are likely to be in one block; 1 access = 12 ms.

5. ... , index not in memory

   Probably need 200 value-pointer pairs for sparse index. If pointer size is 4 bytes and date size is 8, 512/12 = 42.6, so say 40 pairs per block and thus 5 blocks (contiguous). Thus, 1 access for index + 1 access for record = 24 ms.

---

Cost Estimation – Selection Example

Assume: 10,000 records, 50/block

σ_{ArrivalDate} = 15 May

6. sequential file, not contiguous, not sorted (on ArrivalDate), but with secondary index in memory on ArrivalDate

   If we estimate, with the help of someone who knows the application, that the records we need are in 20 different blocks, we have 20 accesses = 240 ms.

7. ... , contiguous, ...

   For 20 blocks, we have 1 seek + 20 rotational delays = 8 ms + 20 \times 4 ms = 88 ms.
**Cost Estimation – Join Example**

\[ \pi_{\text{City}} \sigma_{\text{ArrivalDate} = 15 \text{ May}} (s \times g) \]

Assume: 10,000 @ 50/block for s and 10,000 @ 25/block for g.
Assume: sequential files, not contiguous, g is sorted and has an in-memory index on GuestNr.

1. simple nested-loop join (index not used).
   For each block of s (200) access each block of g; 200 + \(200 \times 400 = 80,200\) accesses = 962,400 ms = 16 min

2. for each block of s, use the index to access blocks of g
   At most 50 blocks of g for each block of s; likely less, say 20; then 200 + 200 \(\times 20 = 4,200\) accesses = 50,400 ms = 50.4 sec.

**Cost Estimation – Join Example**

\[ \pi_{\text{City}} (\sigma_{\text{ArrivalDate} = 15 \text{ May}} S \times g) \quad \text{(equivalent but optimized)} \]

Assume: 10,000 @ 50/block for s and 10,000 @ 25/block for g.
Assume: sequential files, not contiguous, g is sorted and has an in-memory index on GuestNr.

3. Get 15-May records from s and do simple nested-loop join (index not used).
   Access each s block once to get 15-May GuestNr’s and then access each g block once; 200 + 400 = 600 accesses @ 12 ms = 7,200 ms = 7.2 sec

4. Get 15-May records from s and use index to do join.
   Assume we access, say, 10% of the blocks of g using the index; 200 + 40 = 240 accesses or 2,880 ms = 2.9 sec.
Transaction

- Program unit that accesses the database
- Executes atomically (all or nothing) and transforms the DB from one consistent state to another
- Solves problems:
  - system crashes
  - integrity constraint violations
  - concurrent updates

Transaction State Diagram

- Active: All statements executed
- Partially Committed: Able to guarantee completion
- Failed: Unable to guarantee completion
- Committed: Necessary adjustments made (e.g., rollback)
- Aborted: Transaction abort or system crash
Crash Recovery

Logging with Immediate Updates

<T, starts>
<T, file, field, key, old value, new value>
<T, commits>

Can update anytime after log records with old/new values have been written to disk.

Crash Recovery Example

1. Assume the system crashes before the commit as follows:
   <T1, starts>
   <T1, Room, Cost, 1, 90, 95>    assume actual update done
   <T1, Room, Cost, 2, 80, 85>    assume log record on disk
   but update not done
   <T1, Room, Cost, 3, 80, 85>    assume log record not on disk
   <T1, commits>

   Do “undo” for each value record on disk -- overwrites changed or unchanged values.

2. Assume the system crashes after the commit (all log records on disk, but not all updates done). “Redo” each value record; overwrite changed and unchanged values.
Multiple Transactions

- Recovery
  - order serially
  - redo those committed in serial order
  - undo those not committed in reverse serial order

- Checkpoints
  - flush all records from buffer
  - do all updates for committed transactions
  - add <checkpoint> to log
  - subsequently ignore committed transactions before the <checkpoint>

Concurrent Updates

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read # enrolled in CS1</td>
<td>read # enrolled in CS1</td>
</tr>
<tr>
<td>add 1</td>
<td>add 1</td>
</tr>
<tr>
<td>write # enrolled</td>
<td>write # enrolled</td>
</tr>
</tbody>
</table>

If 10 are initially enrolled, the DB says 11 after these transactions execute, but should say 12.
Two-Phase Locking Protocol

- Strict serial execution (works, but allows no concurrency)
- 2PLP (equivalent to serial execution, but allows concurrency)
  - Phase 1: locking – must not unlock
  - Phase 2: unlocking – must not lock

Any attempt by T2 to lock or read N here fails.

T3 can overlap either T1 or T2 or both

Simple Locking is Not Enough

Let A = B = 2 initially.

Execute as written:
- A = 6  B = -4

Execute T1 then T2:
- A = -3  B = -1

Execute T2 then T1:
- A = 5  B = -3

Execution as written is not the same as either serial execution.
Deadlock

T1
LX(A)

Try LX(B)

T2
LX(B)

Try LX(A)

Abort either T1 or T2.

Shared Locks Allow Additional Concurrency

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS(A)</td>
<td>LX(B)</td>
<td>LS(A)</td>
</tr>
<tr>
<td>LS(B)</td>
<td>LS(A)</td>
<td>LS(B)</td>
</tr>
<tr>
<td>UN(B)</td>
<td>UN(A)</td>
<td>UN(B)</td>
</tr>
</tbody>
</table>

LS: read, but not write

 Serializable:

T2

T1

T3