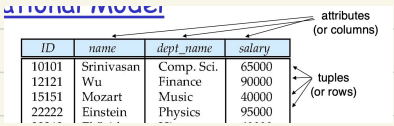
Database

Relational Database Management System (RDBMS)

A Data Model: Underlying the structure of database is a data model

Relational Model: Relation Schema (or Table Format)
Relation (or Relation Instance, Table Content)
Attribute (or Column)
Tuple (or Record, Row)



Relational Algebra: to manipulate set

Relational Operators →

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{salary>=85000}(instructor)$
	Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID,\ salary}(instructor)$
	Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
⋈ (Natural Join)	$instructor \bowtie department$
	Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
× (Cartesian Product)	$instructor \times department$
	Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
∪ (Union)	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$
	Output the union of tuples from the two input relations.

Selection of Tuples

■ Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

■ Select tuples with A=B and D > 5
- $\sigma_{A=B \text{ and } D>5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Selection of Attributes

■ Relation r

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

■ Select A and C
- Projection
- $\Pi_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Joining Two Relations = Cartesian Product

■ Relations r, s:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

■ r x s:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Union of two relations

■ Relations r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

■ r ∪ s:

A	B
α	1
α	2
β	1
β	3

Set difference of two relations

■ Relations r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

■ r − s:

A	B
α	1
β	1

Set Intersection of Two Relations

- Relation r, s:

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

- $r \cap s$

A	B
α	2

Natural Join Example

- Relations r, s:

r

	A	B	C	D
t_r	α	1	α	a
	β	2	γ	a
	γ	4	β	b
	α	1	γ	a
	δ	2	β	b

s

	B	D	E
t_s	1	a	α
	3	a	β
	1	a	γ
	2	b	δ
	3	b	ε

- Natural Join
  - r ⋈ s

	A	B	C	D	E
t	α	1	α	a	α
	α	1	α	a	γ
	α	1	γ	a	α
	α	1	γ	a	γ
	δ	2	β	b	δ

同名的 attributes 下有相同 value 的 row 才会被 join 到一起

Let r and s be relations on schemas R and S respectively. Then, the "natural join" of relations, R and S, is a relation on schema $R \cup S$ obtained as follows:

- Consider every pair of tuples t_r from r and t_s from s.
- If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
  - t has the same value as t_r on r
  - t has the same value as t_s on s

Relational Algebra II

Projection Operation

Can specify functions on attributes

Can assign names to computed values

$\Pi_{F_1, F_2 \ldots F_n}(E)$

F_i : arithmetic expressions
E : expression that produces a relation
Can also name values : F_i as name
Can use to provide derived attributes

Example:

"Compute available credit for every credit account."

$\Pi_{cred\_id,\ (limit - balance)\ \textbf{as}\ available\_credit}(credit\_acct)$

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

⇨

cred_id	available_credit
C-273	2350
C-291	150
C-304	11500
C-313	275

Aggregate Functions

a collection of values → a single result

sum / avg / count / min / max

work on multisets, NOT sets (a value can appear in the input multiple times)

Example:

"Find the total amount owed to the credit company."

$\mathcal{G}_{\textbf{sum}(balance)}(credit\_acct)$

4275

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

"Find the maximum available credit of any account."

$\mathcal{G}_{\textbf{max}(available\_credit)}(\Pi_{(limit - balance)\ \textbf{as}\ available\_credit}(credit\_acct))$

11500

Grouping and Aggregation — compute aggregates on a per-item basis

Example

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

"How many puzzles has each person completed?"

$_{person\_name}\mathcal{G}_{\textbf{count}(puzzle\_name)}(completed)$

- First, input relation *completed* is grouped by unique values of *person_name*
- Then, **count**(*puzzle_name*) is applied separately to each group

$_{person\_name}\mathcal{G}_{\textbf{count}(puzzle\_name)}(completed)$

Input relation is grouped by *person_name*

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Alex	puzzle box
Bob	puzzle box
Bob	soma cube
Carl	altekruse
Carl	puzzle box
Carl	soma cube

⇨

Aggregate function is applied to each group

person_name	
Alex	3
Bob	2
Carl	3

Distinct Values — compute aggregates over sets of values, instead of multisets

Example

How many puzzles has each person completed?

- Using *completed_times* relation this time

person_name	puzzle_name	seconds
Alex	altekruse	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruse	285
Bob	puzzle box	215
Alex	altekruse	290

completed_times

- Each puzzle appears multiple times now.

person_name	puzzle_name	seconds
Alex	altekruse	350
Alex	soma cube	45
Bob	puzzle box	240
Carl	altekruse	285
Bob	puzzle box	215
Alex	altekruse	290

completed_times

- Need to count distinct occurrences of each puzzle's name

$_{person\_name}\mathcal{G}_{\textbf{count-distinct}(puzzle\_name)}(completed\_times)$

append -distinct to any aggregate function to specify elimination of duplicates

e.g: count , count-distinct

⊗ min max

General Form of Aggregates

$G_1, G_2 \dots G_n\ \mathcal{G}\ F_1(A_1), F_2(A_2), \dots, F_m(A_m)\ (E)$

E : expression that provides a relation

Gi : attributes of E to group on . if no Gi specified, then no grouping performed (one big group)

Fi : aggregate function applied to Attribute Ai of E

1st input relation is divided into groups

2nd aggregate functions applied to each group

Fi(Ai) attributes are unnamed! Fi(Ai) as attr_name

Another Example

puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

"How many people have completed each puzzle?"

$_{puzzle\_name}\mathcal{G}_{count(person\_name)}(completed)$

- What if nobody has tried a particular puzzle?
  - Won't appear in *completed* relation

puzzle_name
altekruse
soma cube
puzzle box
clutch box

puzzle_list

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- New puzzle added to *puzzle_list* relation
  - Would like to see { "clutch box", 0 } in result...
  - "clutch box" won't appear in result!
- Joining the two tables doesn't help either
  - Natural join won't produce any rows with "clutch box"

用 outer join operation 解决这个问题

Outer Joins

Natural join requires that both left and right tables have a matching tuple

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \ldots \wedge r.A_n=s.A_n}(r \times s))$$

Outer Join: designed to handle missing information

missing information is represented by null values
null = unknown or unspecified value

Forms of Outer Join

Left outer join: $r \mathbin{⟕} s$
Right outer join: $r \mathbin{⟖} s$
Full outer join: $r \mathbin{⟗} s$

- Summary:

$r =$

attr1	attr2
a	r1
b	r2
c	r3

$s =$

attr1	attr3
b	s2
c	s3
d	s4

$r \bowtie s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3

$r \mathbin{⟕} s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3

$r \mathbin{⟖} s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	*null*	s4

$r \mathbin{⟗} s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3
d	*null*	s4

= 朝着哪边，哪边的带有 null 的 tuple 会被保留

Effects of null values

Arithmetic operations (+, −, *, /) involving null always evaluate to null
e.g. 5 + null = null

Comparison operations involving null evaluates to unknown
null = null evaluates unknown

Boolean Operators and unknown

- and
  - true ∧ *unknown* = *unknown*
  - false ∧ *unknown* = false
  - *unknown* ∧ *unknown* = *unknown*
- or
  - true ∨ *unknown* = true
  - false ∨ *unknown* = *unknown*
  - *unknown* ∨ *unknown* = *unknown*
- not
  - ¬ *unknown* = *unknown*

Relational Operations

For each relational operation, need to specify behavior with respect to null & unknown

select $\sigma_p(E)$

if p evaluates to unknown for a tuple, that tuple is excluded from the result

natural join $r \bowtie s$

a Cartesian product, then a select

If a common attribute has a null value, tuples are excluded from join result

Query Language and Query Processing

SQL: A Query Language

e.g.(1) Find the name of the instructor with ID 11111

```
select name
from instructure
where instructure.ID = 11111
```

e.g.(2) Find all instructor names if they are in a department whose budget is greater than 100,000

```
selece instructor.name , department.dept_name
from instructor, department
where instructor.dept_name = department.dept_name
    and department.budget >= 100,000
```

From SQL to Relational Algebra ↓

$\Pi_{(name,\ dept\_name)}$

|

$\sigma_{(budget\ >\ 100,000)}$

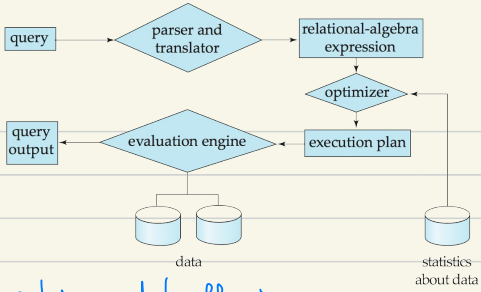
|

$\bowtie$

instructor department

From Relational Operators to Algorithms ↓

```
foreach tuple t in the relation instructor
    foreach tuple s in the relation department
        if (t.dept-name = s.dept_name)
            join the two tuples t and s as a tuple T
            store tuple T in a temporal relation
```



Query Processing

Query parsing and optimization

Query translation

Query evaluation using indexing / hashing and buffering

Parallel Database & SQL or NoSQL 略

Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

Basic Concepts

Indexing mechanisms used to speed up access to desired data
(e.g., author catalog in library)

Search key: attribute to set of attributes used to look up records in a file

An index file consists of records (called index entries) of the form

search-key	pointer

Index files are typically much smaller than the original file

Two basic kinds of indices:
Ordered indices: search keys are stored in sorted order
Hashing indices: search keys are distributed uniformly across "buckets" using a "hash function"

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

index entries are sorted stored on the search key

Primary Index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file

Also called clustering index

Search key of a primary index is usually but not necessary the primary key.

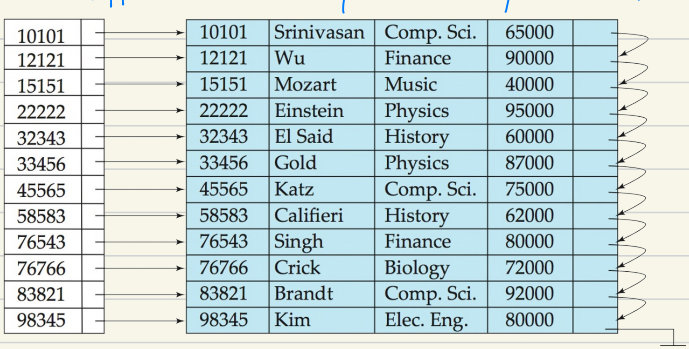
Secondary Index: an index whose search key specifies an order different from the sequential order of the file.

Also called non-clustering index.

Index-sequential file: ordered sequential file with a primary key

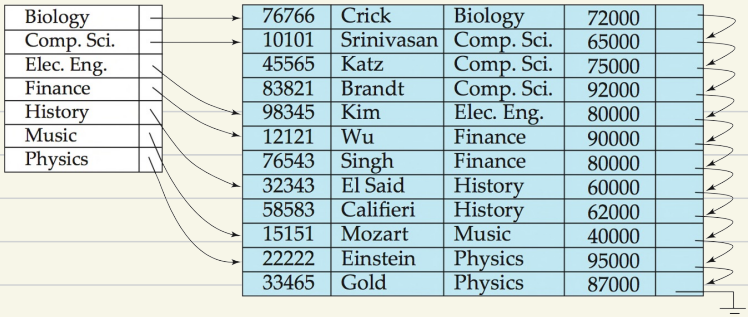
Dense Index Files

Dense index: Index record appears for every search-key value in the file



10101	10101	Srinivasan	Comp. Sci.	65000	
12121	12121	Wu	Finance	90000	
15151	15151	Mozart	Music	40000	
22222	22222	Einstein	Physics	95000	
32343	32343	El Said	History	60000	
33456	33456	Gold	Physics	87000	
45565	45565	Katz	Comp. Sci.	75000	
58583	58583	Califieri	History	62000	
76543	76543	Singh	Finance	80000	
76766	76766	Crick	Biology	72000	
83821	83821	Brandt	Comp. Sci.	92000	
98345	98345	Kim	Elec. Eng.	80000	

Dense index on *dept_name*, with instructor file sorted on *dept_name*



Sparse Index Files

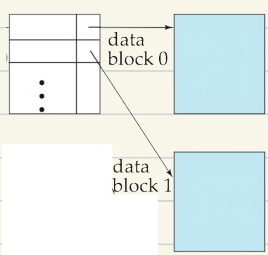
Sparse Index : contains index records for only some index-key values

Applicable when records are sequentially ordered on search-key

To locate a record with search-key value K:
- Find index record with largest search-key value < K
- Search file sequentially starting at the record to which the index record points

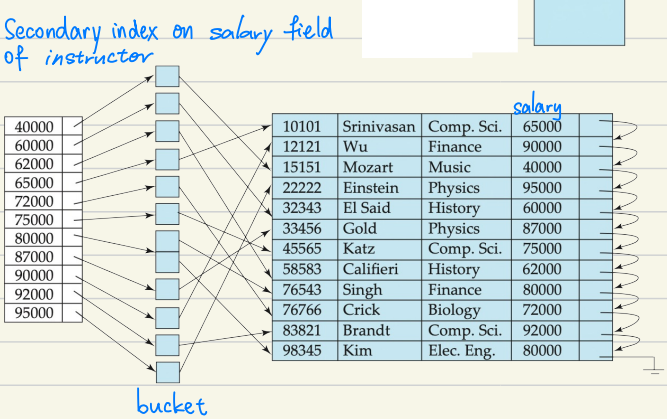
Compared to dense indices:
- Less space and less maintenance overhead for insertions and deletions
- Generally slower than dense index for locating records

Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Secondary Indices Example

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification —— when a file is modified, every index on the file must be updated.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

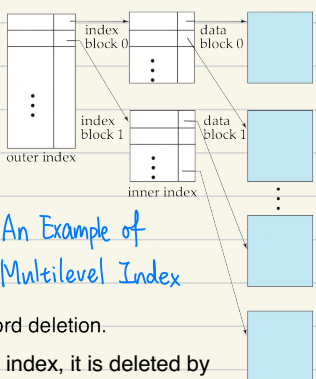
Multilevel Index 解决 index 过大，无法放入内存的问题

If primary index does not fit in memory, access becomes expensive.

Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
- outer index – a sparse index of primary index
- inner index – the primary index file

If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

Indices at all levels must be updated on insertion or deletion from the file.



An Example of Multilevel Index

Index Update: Deletion

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index entry deletion:

- Dense indices
  - deletion of search-key is similar to file record deletion.
- Sparse indices
  - If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
  - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

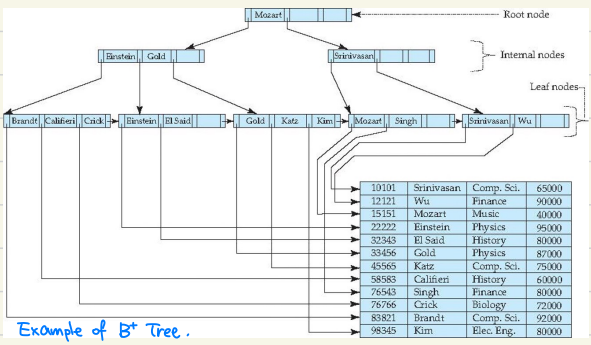
Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

B^+-Tree Index Files

An alternative to index-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively



Example of B^+ Tree.

B^+-Tree Properties:

All paths from root to leaf are of the same length

Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children

A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values

Special cases:

If the root is not a leaf, it has at least 2 children

If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values

B^+-Tree Node Structure

Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

K_i are the search-key values

P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

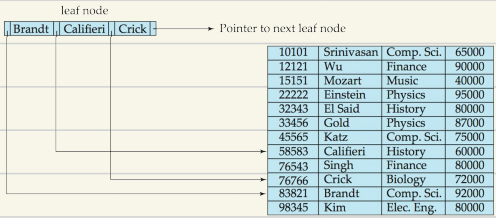
Search-keys in a node are ordered: $K_1 < K_2 < K_3 < \ldots < K_{n-1}$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B^+-Trees

Properties of a leaf node

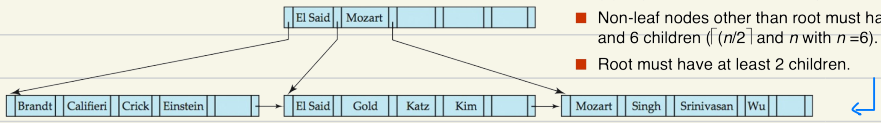
- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



Non-Leaf Nodes in B^+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with n pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \le i \le n-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

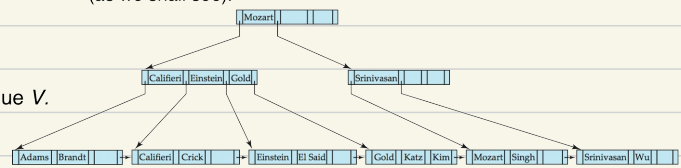
- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.



B^+-tree for instructor file ($n = 6$)

Observations about B^+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2^* \lceil n/2 \rceil$ values
  - Next level has at least $2^* \lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).



Queries on B^+-Trees

- Find record with search-key value V.
  1. C=root
  2. While C is not a leaf node {
     1. Let i be least value s.t. $V \leq K_i$.
     2. If no such exists, set C = *last non-null pointer in C*
     3. Else { if (V= K_i) Set C = P_{i+1} else set $C = P_i$}

     }
  3. Let i be least value s.t. $K_i = V$
  4. If there is such a value i, follow pointer P_i to the desired record.
  5. Else no record with search-key value k exists.

Handling Duplicates

- With duplicate search keys
  - In both leaf and internal nodes,
    - we cannot guarantee that $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
    - but can guarantee $K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n-1}$
  - Search-keys in the subtree to which P_i points
    - are $\leq K_i$, but not necessarily $< K_i$,
    - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V
- We modify find procedure as follows
  - traverse P_i even if $V = K_i$
  - As soon as we reach a leaf node C check if C has only search key values less than V
    - if so set C = right sibling of C before checking whether C contains V
- Procedure printAll
  - uses modified find procedure to find first occurrence of V
  - Traverse through consecutive leaves to find all occurrences of V

Queries on B+-Trees (cont'd)

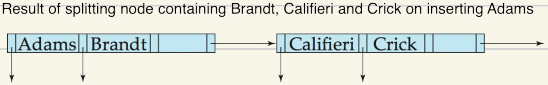
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B+-Trees : Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   1. Add record to the file
   2. If necessary, add a pointer to the bucket.
3. If the search-key value is not present, then
   1. Add the record to the main file (and create a bucket if necessary)
   2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the following.

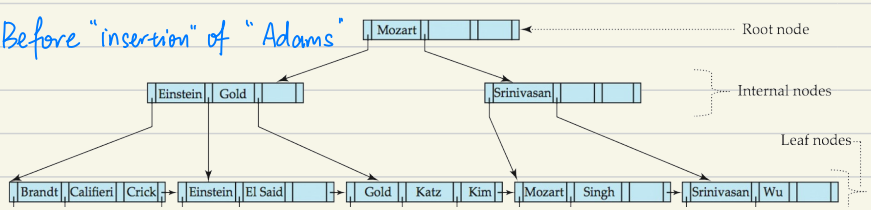
- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

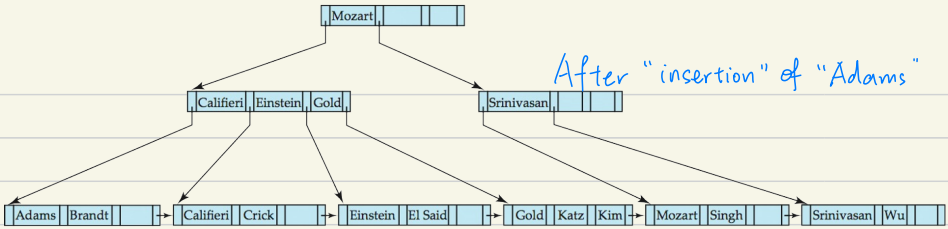
Example Result of splitting node containing Brandt, Califieri and Crick on inserting Adams



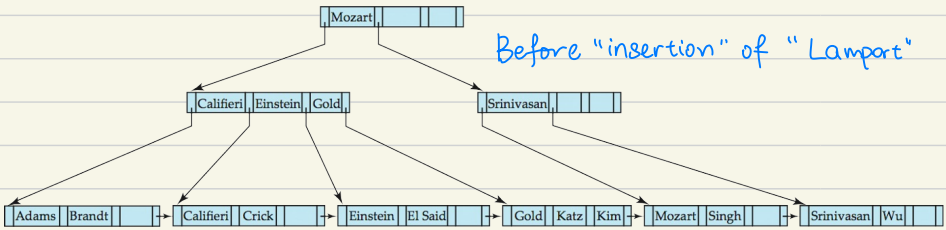
Next step : insert entry with (Califieri, pointer-to-new-node) into parent

Before "insertion" of "Adams"



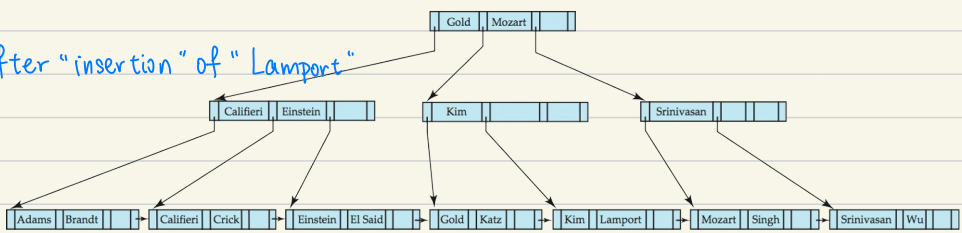


After "insertion" of "Adams"



Before "insertion" of "Lamport"

After "insertion" of "Lamport"



Insertion in B+- Trees

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

Updates on B+- Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

B^+-Tree File Organization

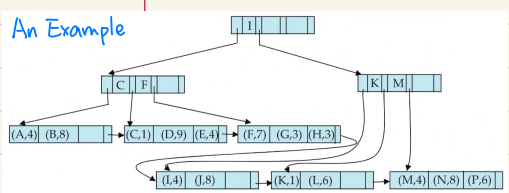
- Index file degradation problem is solved by using B^+-Tree indices.
- Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.

Leaf nodes are still required to be half full

- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.

Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

An Example



Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000
- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
  3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1 = b_1$ and $a_2 < b_2$

Indices on Multiple Attributes

Suppose we have an index on combined search-key (dept-name, salary)

- With the **where** clause
  where *dept_name* = "Finance" **and** *salary* = 80000
  the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.
  - Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
  where *dept_name* < "Finance" **and** *balance* = 80000
  - May fetch many records that satisfy the first but not the second condition

Hashing

Static Hashing

A bucket is a unit of storage containing one or more records (a bucket is typically a disk pool).

In a hash file organization, we obtain the bucket of a record directly from its search-key value using a hash function.

Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B

Hash function is used to locate records for access, insertion as well as deletion

Records with different search-key values may be mapped to the same bucket; thus, entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization:

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

Handling of Bucket Overflows

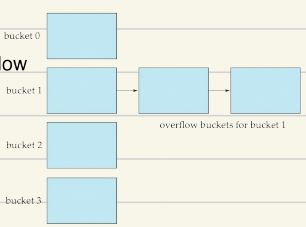
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

hash func 理想状态是 uniformly distribute

- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets.***

解决 Overflow buckets 的方法 = Overflow chaining { closed hashing / open hashing (不建议)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing.**
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



Hash Indices

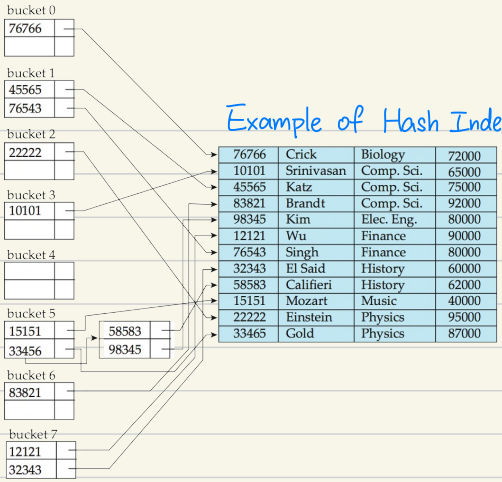
Hash can also be used for index-structure creation.

Hash index: organizes the search keys, with their associated record pointers, into a hash file structure

Hash indices are always secondary indices

If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.

However, we use the term hash index to refer to both secondary index structures and hash organized files.



Example of Hash Index

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

Deficiencies of Static Hashing

In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.

If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
If database shrinks, again space will be wasted.

One solution: periodic re-organization of the file with a new hash function

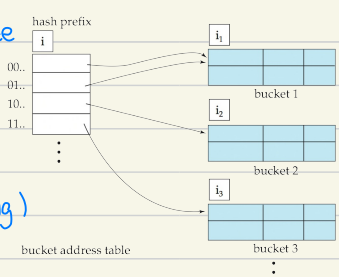
Expensive, disrupts normal operations

Better solution: allow the number of buckets to be modified dynamically

Dynamic Hashing

Good for database that grows and shrinks in size

Allows the hash function to be modified dynamically



Extendable Hashing (one form for dynamic hashing)

Hash function generates values over a large range, typically b-bit integers, $b = 32$

At any time use only a prefix of the hash function to index into a table of bucket addresses

Let the length of the prefix be i bits, $0 \le i \le 32$

Bucket address table size = 2^i. Initially $i = 0$
Value of i grows and shrinks as the size of the database grows and shrinks

Multiple entries in the bucket address table may point to a bucket

Thus, actual number of buckets is $< 2^i$
The number of buckets also changes dynamically due to coalescing and splitting of buckets.

Use of Extendable Hash Structure

Each bucket j stores a value i_j

- All the entries that point to the same bucket have the same values on the first i_j bits.

To locate the bucket containing search-key K_j,

1. Compute $h(K_j) = X$
2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket

To insert a record with search-key value K_j,

- Follow same procedure as look-up and locate the bucket, say j.
- If there is room in the bucket j insert record in the bucket.
- Else the bucket must be split and insertion re-attempted (next slide.)
  - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure

To split a bucket j when inserting record with search-key value K_j

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

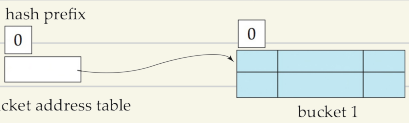
Deletion in Extendable Hash Structure

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

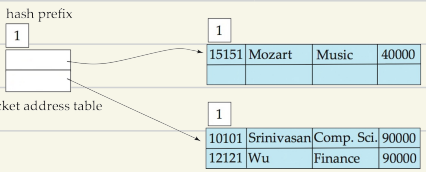
Example of Extendable Hash Structure

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

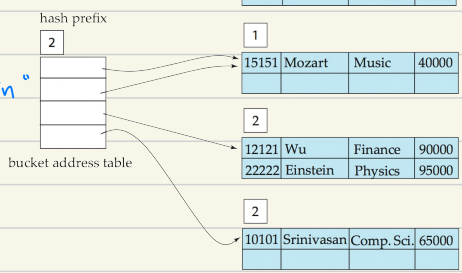
Initial Hash Structure ; bucket size =2



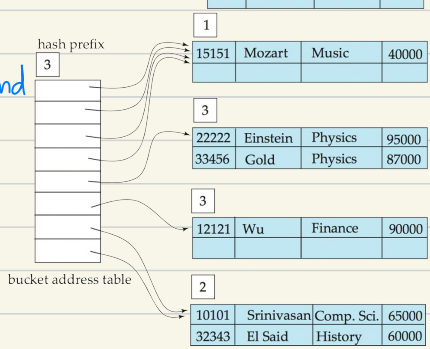
Hash structure after insertion of "Mozart", "Srinivasan" and "Wu" records



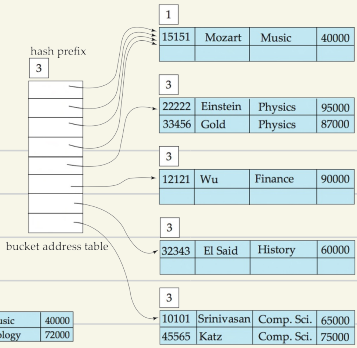
Hash structure after insertion of "Einstein" record



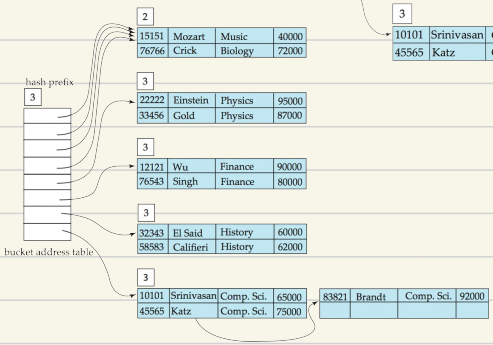
Hash structure after insertion of "Gold" and "El Said" records



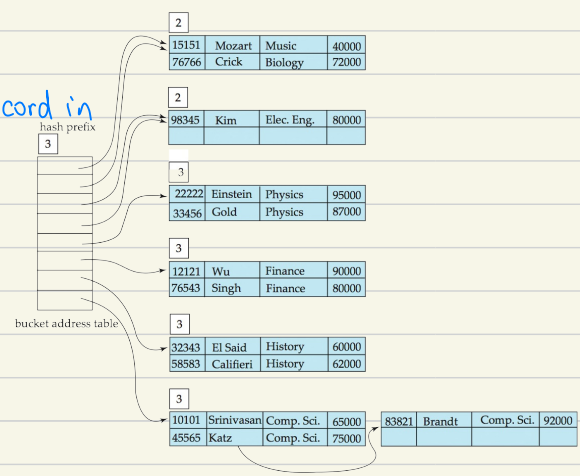
Hash structure after insertion of "Katz" record



After insertion of 11 records



After insertion of "Kim" record in previous hash structure



Extendable Hashing vs. Other Schemes

Benefits of extendable hashing

- Hash performance does not degrade with growth of file
- Minimal space overhead

Disadvantages of extendable hashing

- Extra level of indirection to find desired record
- Bucket address table may itself become very big (larger than memory)
  - Cannot allocate very large contiguous areas on disk either
  - Solution: B$^+$-tree structure to locate desired record in bucket address table
- Changing size of bucket address table is an expensive operation

Linear Hashing is an alternative mechanism

- Allows incremental growth of its directory (equivalent to bucket address table)
- At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

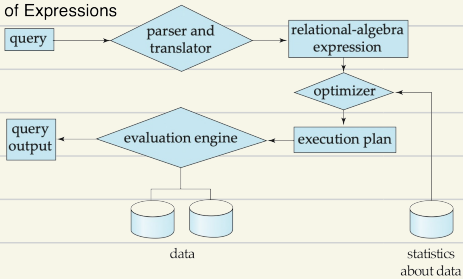
- Allows incremental growth of its directory (equivalent to bucket address table)
- At the cost of more bucket overflows

Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Parsing and translation

Translate the query into its internal form. This is then translated into relational algebra.
Parser checks syntax, verifies relations.

Evaluation

The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

✡ Optimization

A relational algebra expression may have many equivalent expressions

- E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

Each relational algebra operation can be evaluated using one of several different algorithms

Annotated expression specifying detailed evaluation strategy is called an evaluation-plan

- E.g., can use an index on *salary* to find instructors with salary < 75000,
- or can perform complete relation scan and discard instructors with salary ≥ 75000

Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.

Cost is estimated using statistical information from the database catalog

- e.g. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

Cost is generally measured as total elapsed time for answering query

Typically: disk access, predominate cost, easy to estimate

number of seeks	average-seek-cost
number of blocks read	average-block-read-cost
number of blocks written	average-block-write-cost

Cost to write is greater than cost to read

data is read back after being written to ensure that the write was successful

Just use the number of block transfers from disk and the number of seeks as the cost measures.

t_T - time to transfer one block

t_S - time for one seek

Cost for b block transfers plus S seeks : $b * t_T + S * t_S$

Do not include cost to writing output to disk in our cost formulae

Selection Operation

File scan

Algorithm A1 (linear search). Scan each file block and test all records to see whether they satisfy selection condition.

Cost estimate = b_r block transfers + 1 seek

b_r denotes number of blocks containing records from relation r

If selection is on a key attribute, can stop on finding record

cost = $(b_r/2)$ block transfers + 1 seek

Linear search can be applied regardless of
selection condition or
ordering of records in the file, or
availability of indices

Note: binary search generally does not make sense since data is not stored consecutively except when there is an index available,
and binary search requires more seeks than index search

Selection Using Indices

Index Scan – search algorithms that use an index
selection condition must be on search-key of index

A2 (primary index, equality on key) Retrieve a single record that satisfies the corresponding equality condition

Cost = $(h_i + 1) * (t_T + t_S)$ where h_i is the height of the ith B+ Tree index

A3 (primary index, equality on non-key) Retrieve multiple records

Records will be on consecutive blocks
Let b = number of blocks containing matching records

Cost = $h_i * (t_T + t_S) + t_S + t_T * b$

A4 (secondary index, equality on non-key)

Retrieve a single record if the search-key is a candidate key

Cost = $(h_i + 1) * (t_T + t_S)$

Retrieve multiple records if search-key is not a candidate key

Cost = $(h_i + n) * (t_T + t_S)$

Very expensive.

Selections Involving Comparison

Can implement selections of the form $\sigma_{A \leq v}(r)$ or $\sigma_{A \geq v}(r)$ by using
a linear file scan or
by using indices in the following ways:

A5 (primary index, comparison). (Relation is sorted on A)

For $\sigma_{A \geq v}(r)$ use index to find first tuple $\geq v$ and scan sequentially from there

For $\sigma_{A \leq v}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

A6 (secondary index, comparison)

For $\sigma_{A \geq v}(r)$, use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.

For $\sigma_{A \leq v}(r)$, just scan leaf pages of index finding pointers to records, till first entry $> v$

In either case, retrieve records that are pointed to

requires an I/O for each record

linear file scan might be cheaper

Algorithm	conditions	operator(s)	attributes	cost
A1 (linear)	None	$=$	key	$b_r/2 \cdot t_T + t_S$
		other ops	any	$b_r \cdot t_T + t_S$
A2 (index)	primary index	$=$	key	$(h_i + 1)(t_T + t_S)$
A3 (index)	primary index	$=$	non-key	$h_i(t_T + t_S) + t_S + b \cdot t_T$
A4 (index)	secondary index	$=$	key	$(h_i + 1)(t_T + t_S)$
			non-key	$(h_i + n)(t_T + t_S)$
A5 (index)	primary index	$\geq$	any	$(h_i(t_T + t_S) + t_S + b \cdot t_T$
A6 (index)	second index	$\geq$	any	$(h_i + n)(t_T + t_S)$

"key" in the 4th column in the table means a primary/candidate key defined when creating a relation.

b_r is the number of blocks for a relation r.

b is the number of blocks containing the records that match a search key.

n is the number of records that match a search key.

Implementation of Complex Selections

Conjunction: $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$

A7 (conjunctive selection using one index)

Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.

Test other conditions on tuple after fetching it into memory buffer

A8 (conjunctive selection using composite index)

Use appropriate composite (multiple-key) index if available

A9 (conjunctive selection by intersection of identifiers)

Requires indices with record pointers.
Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
Then fetch records from file
If some conditions do not have appropriate indices, apply test in memory.

Disjunction: $\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$

Alo (disjunctive selection by union of identifiers)

Applicable if all conditions have available indices.
Otherwise use linear scan.
Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
Then fetch records from file

Negation: $\sigma_{\neg\theta}(r)$
Use linear scan on file

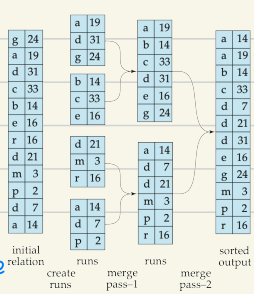
If very few records satisfy $\neg\theta$, and an index is applicable to θ
Find satisfying records using index and fetch from file

Sorting

We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.

For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.

External Sort-Merge
Let M denote memory size (in pages)
1. Create sorted runs. Let i be 0 initially
repeatedly do the following till the end of the relation:
(a) read M blocks of relation into memory
(b) sort the in-memory blocks
(c) write sorted data to run R_i: increment i
let the final value of i be N
2. Merge the runs (N-way merge). We assume (for now) that $N < M$.
1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
2. repeat
1. select the first record (in sort order) among all buffer page
2. write the record to the output buffer. If the output buffer is full write it to disk



3. delete the record from its input buffer page
   if the buffer page becomes empty, then read the next block (if any) of the run into the buffer
3. until all input buffer pages are empty.

If N ≥ M, several merge passes are required

- In each pass, contiguous groups of $M - 1$ runs are merged.
- A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor.
  - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
- Repeated passes are performed till all runs have been merged into one.

Cost Analysis

- 1 block per run leads to too many seeks during merge
  - Instead use b_b buffer blocks per run
    → read/write b_b blocks at a time
  - Can merge $\lfloor M/b_b \rfloor - 1$ runs in one pass
- Total number of merge passes required: $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil$.
- Block transfers for initial run creation as well as in each pass is $2b_r$
  - for final pass, we don't count write cost
    - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
  - Thus total number of block transfers for external sorting:
    $$b_r (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r / M) \rceil + 1)$$

Cost of seeks

- During run generation: one seek to read each run and one seek to write each run
  - $2\lceil b_r / M \rceil$
- During the merge phase
  - Need $2\lceil b_r / b_b \rceil$ seeks for each merge pass
    - except the final one which does not require a write
  - Total number of seeks:
    $$2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r / M) \rceil - 1)$$

Join Operation

several different algorithms to implement joins

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge join
- Hash-join

Choice based on cost estimation

- Examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

Students & takes 分别为两个 relation

- **for each** tuple t_r **in** r **do begin**
  for each tuple t_s **in** s **do begin**
  test pair (t_r, t_s) to see if they satisfy the join condition θ
  if they do, add $t_r \cdot t_s$ to the result.
  end
  end

Nested-Loop Join

To compute the theta join $r \bowtie_\theta s$

r is called the outer relation
s the inner relation of the join

Requires no indices and can be used with any kind of join condition

Expensive: it examines every pair of tuples in the two relations

In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

n - record number
b - block number

$n_r * b_s + b_r$ block transfers, plus
$n_r + b_r$ seeks

If the smaller relation fits entirely in memory, use that as the inner relation.
Reduces cost to $b_r + b_s$ block transfers and 2 seeks.

- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.

Block nested-loops algorithm is preferable

Block Nested-Loop Join

Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation

- **for each** block B_r **of** r **do begin**
  for each block B_s **of** s **do begin**
  for each tuple t_r **in** B_r **do begin**
  for each tuple t_s **in** B_s **do begin**
  Check if (t_r, t_s) satisfy the join condition
  if they do, add $t_r \cdot t_s$ to the result.
  end
  end
  end
  end

Worst case estimate:

$b_r * b_s + b_r$ block transfers + 2 seeks

Each block in the inner relation s is read once for each block in the outer relation

Best case:

$b_r + b_s$ block transfers + 2 seeks

Improvements to nested-loop and block nested-loop algorithms

In block nested-loop, use M-2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use the remaining 2 blocks to buffer inner relation and output

- Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2\lceil b_r / (M-2) \rceil$ seeks

If equi-join attribute forms a key on inner relation, stop inner loop on first match

Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)

Use index on inner relation if available

Indexed Nested-Loop Join

Index lookups can replace file scans if

join is an equi-join or natural join and

an index is available on the inner relation's join attribute

can construct an index just to compute a join

For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.

Worst case:

buffer has space for only one page of r, and, for each tuple in r, we perform an index lookup on s

Cost of the join: $b_r(t_T + t_S) + n_r * c$

where c is the cost of traversing index and fetching all matching s tuples for one tuple or r.

c can be estimated as cost of a single selection on s using the join condition.

If indices are available on join attributes of both *r* and *s*, use the relation with fewer tuples as the outer relation

Example of Nested-Loop Join Costs

- Compute *student* ⋈ *takes*, with *student* as the outer relation.
- Let *takes* have a primary B+-tree index on the attribute *ID*, which contains 20 entries in each index node.
- Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *student* has 5000 tuples
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join $\log_{20/2}(10000) = \log_{10}(10000) = 4$
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join

- Examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

Merge Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes)
2. Merge the sorted relations to join them
   1. Join step is like the merge stage of the sort-merge algorithm
   2. Main difference is handling of duplicate values in join attribute – every pair with same value on join attribute must be matched

pr

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

ps

a1	*a3*
a	A
b	G
c	L
d	N
m	B

s

Only for equi-joins and natural joins

Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)

The cost of merge join is:

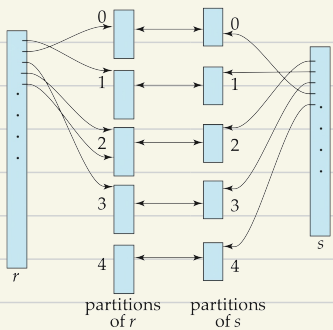
- $b_r + b_s$ block transfers $+\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks + the cost of sorting if relations are unsorted

hybrid merge-join

- **hybrid merge-join**: If one relation is sorted, and the other has a secondary B+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- h maps *JoinAttrs* values to {0, 1, ..., n}, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $r_0, r_1 \ldots, r_n$ denotes partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.



- *Note:* In book, r_i is denoted as H_{ri}, s_i is denoted as H_{si} and n is denoted as n_h.
- r tuples in r_i need only to be compared with s tuples in s_i
  Need not be compared with s tuples in any other partition, since:
  - an r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value i, the r tuple has to be in r_i and the s tuple in s_i.

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If recursive partitioning required:
  - number of passes required for partitioning build relation s to less than M blocks per partition is $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    $2(b_r + b_s)\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil + b_r + b_s$ block transfers +
    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

instructor ⋈ *teaches*

- Assume that memory size is 20 blocks
- $b_{instructor}$= 100 and $b_{teaches}$ = 400.
- *instructor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *teaches* into five partitions,each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
  - 3(100 + 400) = 1500 block transfers + $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ seeks

Hybrid Hash-Join

- Useful when memory sized are relatively large, and the build input is bigger than memory.
- **Main feature of hybrid hash join:**

  Keep the first partition of the build relation in memory.
- E.g. With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
  - Division of memory:
    - The first partition occupies 20 blocks of memory
    - 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- *teaches* is similarly partitioned into five partitions each of size 80
  - the first is used right away for probing, instead of being written out
- Cost of 3(80 + 320) + 20 +80 = 1300 block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M >> \sqrt{b_s}$

Complex Joins

Other Operations & Evaluation of Expressions