DBS Database Systems
Implementing and Optimising Query Languages

Peter Buneman

9 November 2010

Storage and Indexing

Reading: R&G Chapters 8, 9 & 10.1

We typically store data in external (secondary) storage. Why? Becuase:

- Secondary storage is cheaper. £100 buys you 1gb of RAM or 100gb of disk (2006 figures)
- Secondary storage is more stable. It survives power cuts and – with care – system crashes.

Differences between disk and main memory

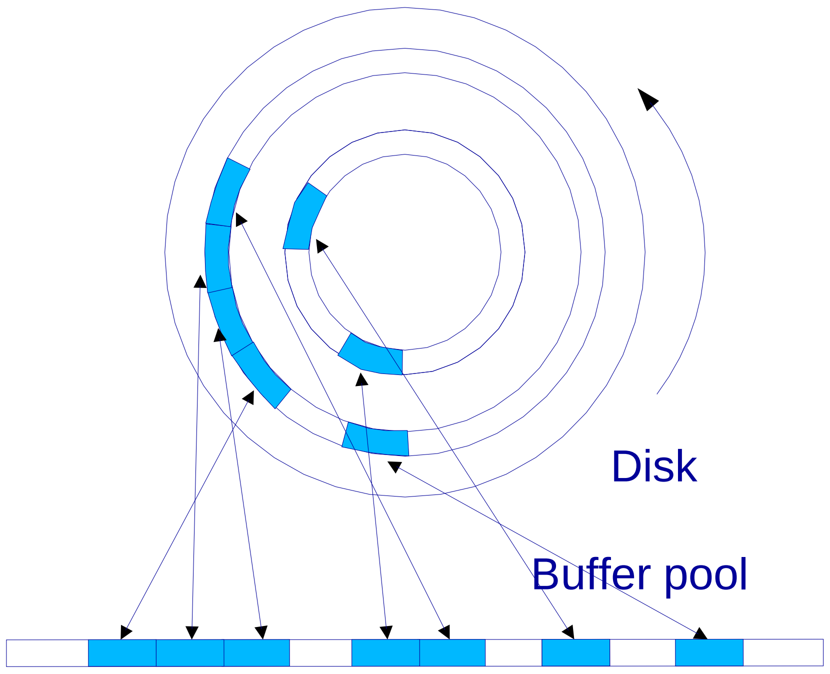
- Smallest retrievable chunk of data: memory $=$ 1 byte, disk $=$ 1 page $=$ 1kbyte (more or less)
- Access time (time to dereference a pointer): memory $< 10^{-8}$ sec, disk $> 10^{-2}$ sec.

However *sequential data*, i.e. data on sequential pages, can be retrieved rapidly from disk.

Communication between disk and main memory

A *buffer pool* keeps images of disk pages in main memory *cache*.

Also needed a table that *maps* between positions on the disk and positions in the cache (in both directions)



When a page is requested

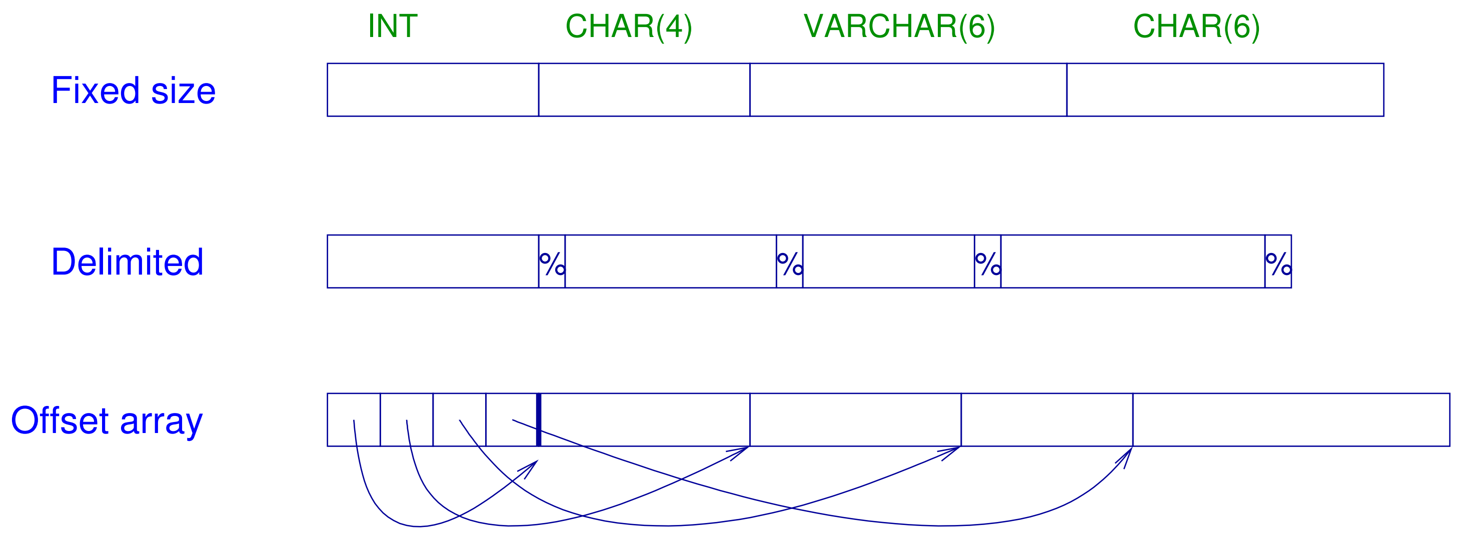
- If page is already in pool present, return address.
- If there is room in the pool, read page in and return address.
- If no room, choose a frame for replacement.
  - if current frame is *dirty* – it has been written to – write frame to disk.
- read page in and return address.

Requesting process may *pin* page. Indicating that it "owns" it.

Page replacement policy: LRU, MRU, random, etc. Pathological examples defeat MRU and LRU.

Storing tuples

Tuples are traditionally stored *contiguoulsy* on disk. Three possible formats (at least) for storing tuples:



Comments on storing tuples

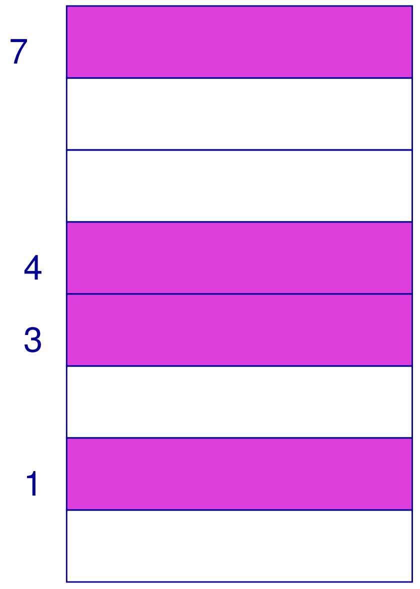
Fixed format appears more efficient. We can "compile in" the offsets. But remember that DB processing is *dominated by i/o*

Delimited can make use of variable length fields (VARCHAR) and simple compression (e.g. deleting trailing blanks)

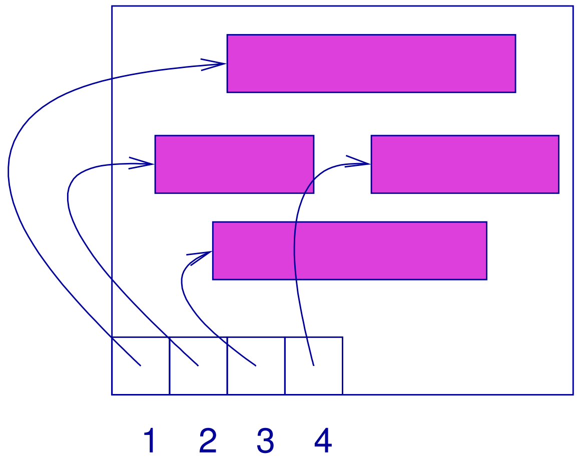
Fixed and delimited formats require *extra* space to represent null values. We get them for free (almost) in the offset array representation.

Placing Records on a Page

We typically want to keep "pointers" or *object identifiers* for tuples. We need them if we are going to build indexes, and we'd like them to be *persistent.*



Array of tuples



Array of pointers

Comments on page layouts

Array of tuples suitable for fixed length records.

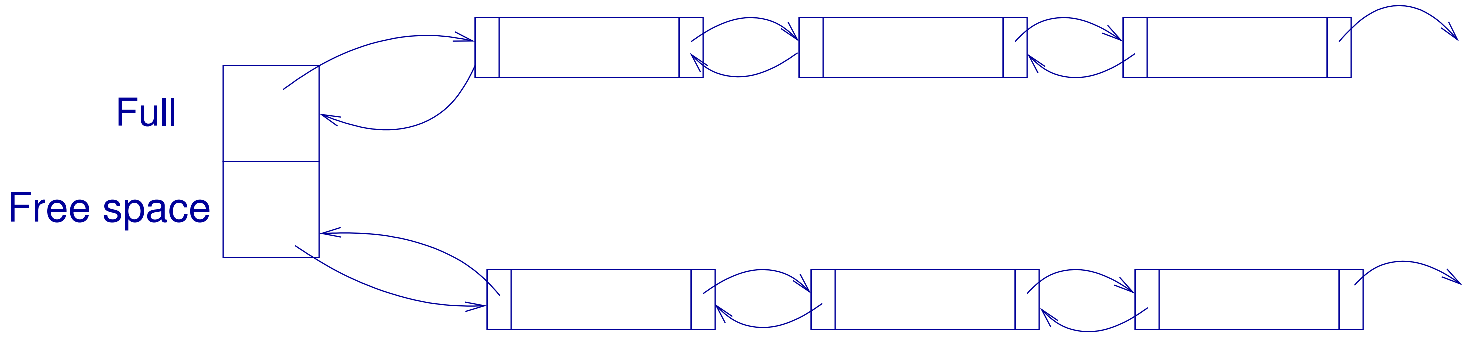
- Object identifier is (page-identifier, index) pair.
- Cannot make use of space economy of variable-length record.

Pointer array is suitable for variable length records

- Object identifier is (page-identifier, pointer-index) pair.
- Can capitalize on variable length records.
- Records may be moved on a page to make way for new (or expanded) records.

File organization – unordered data

Keep two lists: pages with room and pages with no room.



Variations:

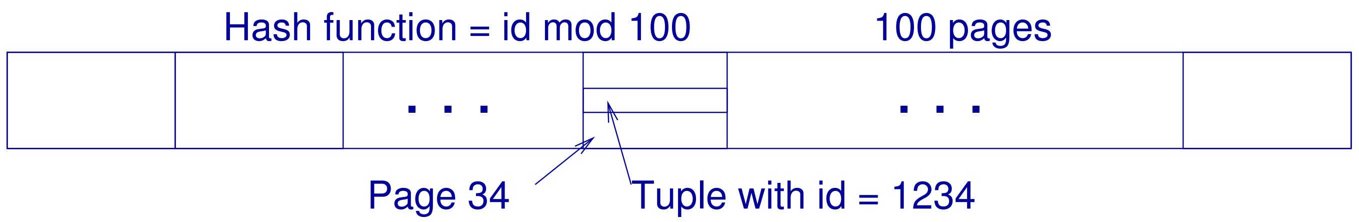
- Keep an *array* of pointers.
- Order by amount of free space (for variable length tuples)

These are called *heap* files.

Other organizations

- Sorted files. Records are kept in order of some attribute (e.g. `Id`). Records are assumed to be fixed-length and "packed" onto pages.

  That is, the file can be treated as an array of records.
- Hashed files. Records are kept in an array of pages indexed by some hash function applied to the attribute. Naive example:



I/O Costs

We are primarily interested in the I/O (number of page reads and writes) needed to perform various operations. Assume B pages and that read or write time is D

	Scan	Eq. Search	Range Search	Insert	Delete
Heap	BD	$0.5BD$	BD	2D	*Search* $+D$
Sorted	BD	$D\log_2 B$	$D\log_2 B + m^*$	*Search* $+BD$	*Search* $+BD$
Hashed	$1.25BD$	D	$1.25BD$	$2D$	*Search* $+D$

* $m =$ number of matches
Assumes 80% occupancy of hashed file

Indexing – Introduction

Index is a collection of *data entries* with efficient methods to locate, insert and delete data entries.

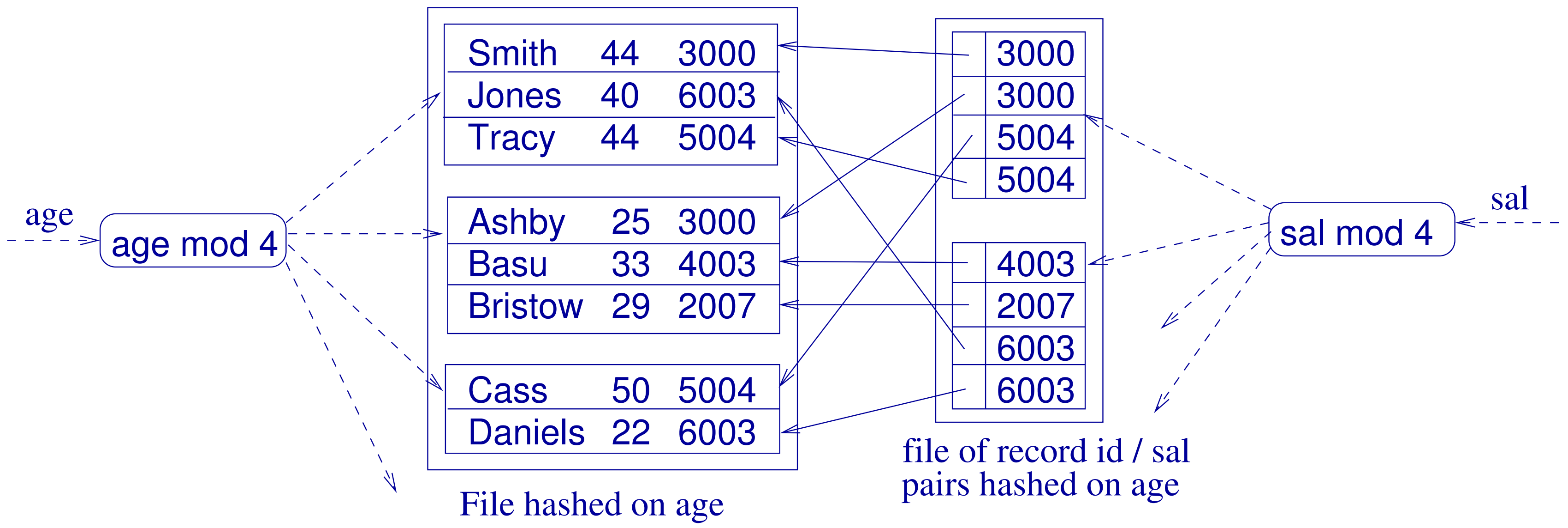
Hashed files and sorted files are simple examples of indexing methods, but they don't do all of these efficiently.

We index on some *key*.

Note the index key is not (necessarily) the "key" in the database design sense of the term.

We can only organize a data file by one key, but we may want indexes on more than one key.

Example. Hash indexes and files



Indexes are needed for optimization

How are these queries helped by the presence of indexes?

```
SELECT   *
FROM     Employee
WHERE    age = 33
```

```
SELECT   *
FROM     Employee
WHERE    age > 33
```

```
SELECT   *
FROM     Employee
WHERE    sal = 3000
```

```
SELECT   *
FROM     Employee
WHERE    sal > 3000
```

What an index can provide

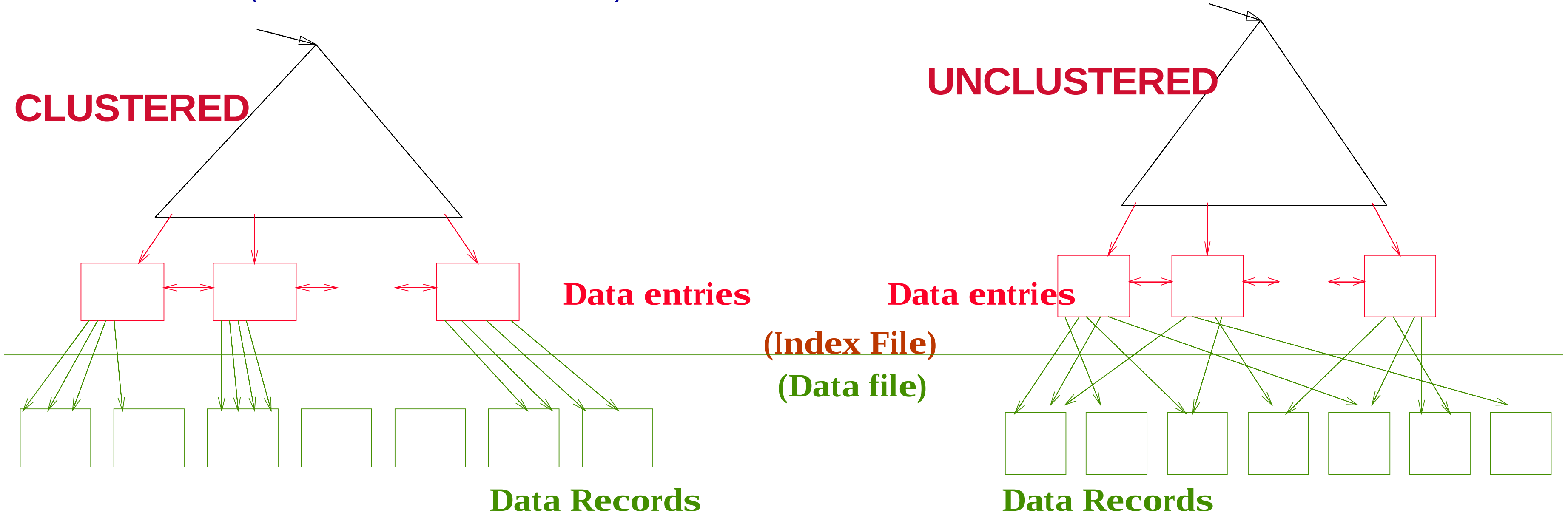
Given a key k, an index returns k^* where k^* is one of three things:

1. A *data record* (the tuple itself) with the search key value k
2. A *pointer* to a record with search key k together with k.
3. A *list of pointers* to records with search key k together with k.

Clustered vs. Unclustered Indexes

If we use tree indexing (to be described) we can exploit the ordering on a key and make *range queries* efficient.

An index is *clustered* if the data entries that are close in this ordering are stored physically close together (i.e. on the same page).



Tree indexing

Why not use the standard search tree indexing techniques that have been developed for main memory data (variations on binary search trees): AVL trees, 3-3 trees, red-black trees, etc?

Reason: binary search is still slow. 10^6 tuples (common) $\log_2(10^6) = 20$ – order 1 second because "dereferencing" a pointer on disk takes between 0.01 and 0.1 seconds.

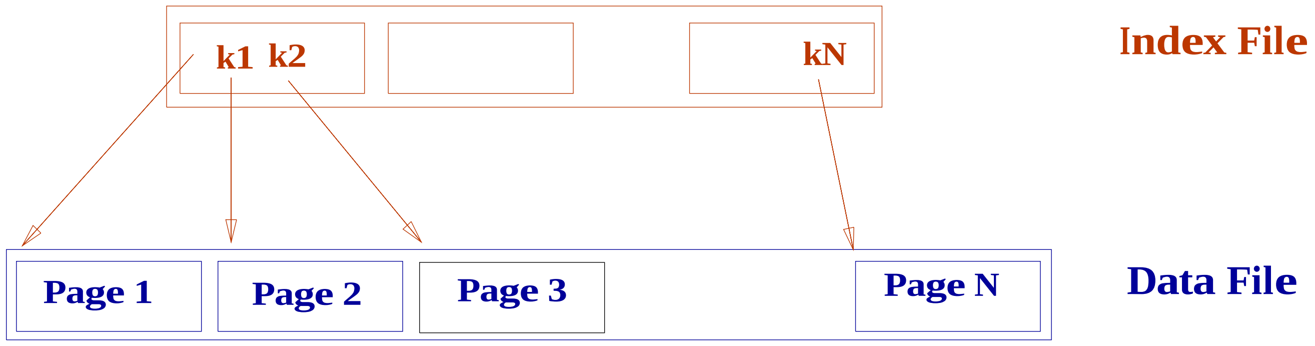
Solution:

1. Use n-ary trees rather than binary.
2. Keep only keys and pointers at internal nodes. Leaves can be data values (either records or record-id/key-value pairs)

Range Search

Example of point (2). We can speed up ordinary binary search on a sorted array by keeping indexes and page pointers in a separate file.

The "index file" will typically fit into cache.



Consider queries such as

```
SELECT   *
FROM     Employee
WHERE    20 < Sal AND Sal < 30
```

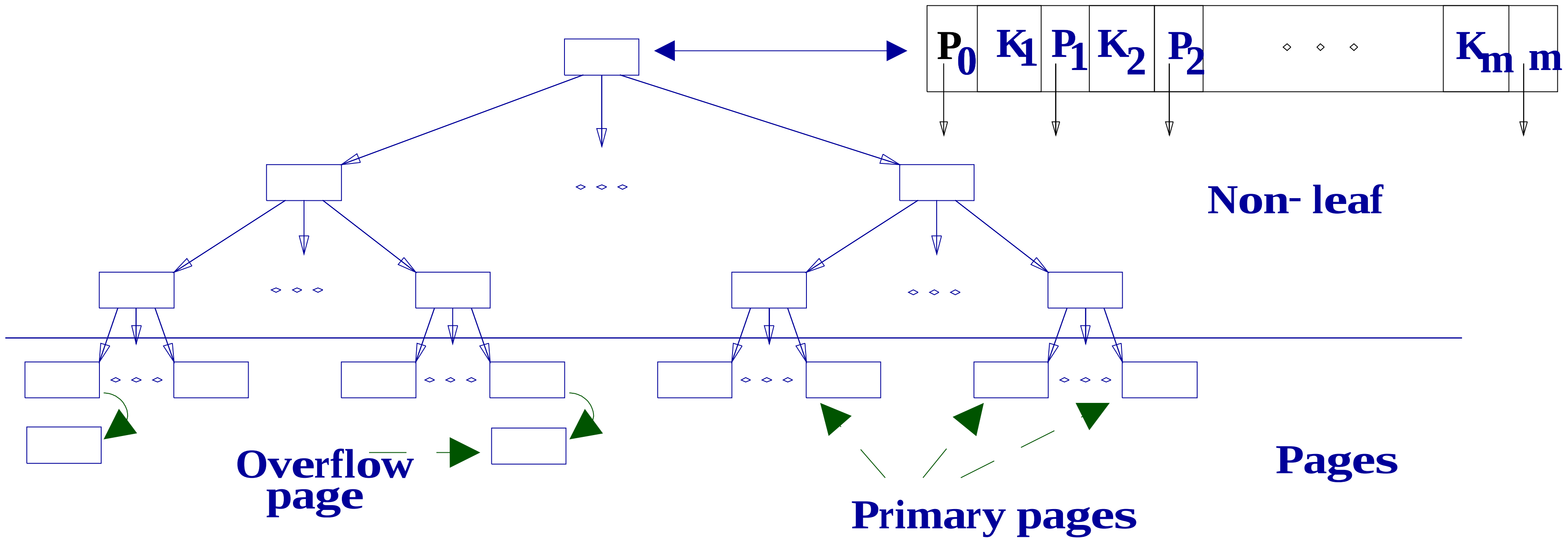
or

"Find all employees whose name begins with 'Mac'." (also a range search)

ISAM

ISAM $=$ *Indexed Sequential Access Method*: a search tree whose nodes contain m keys and $m+1$ pointers. m is chosen to "fill out" a page. A "pointer" is a page-id.

The pointer p_i between keys k_{i-1} and k_i points to a subtree whose keys are all in the range $k_{i-1} < k < k_i$.



How ISAM works

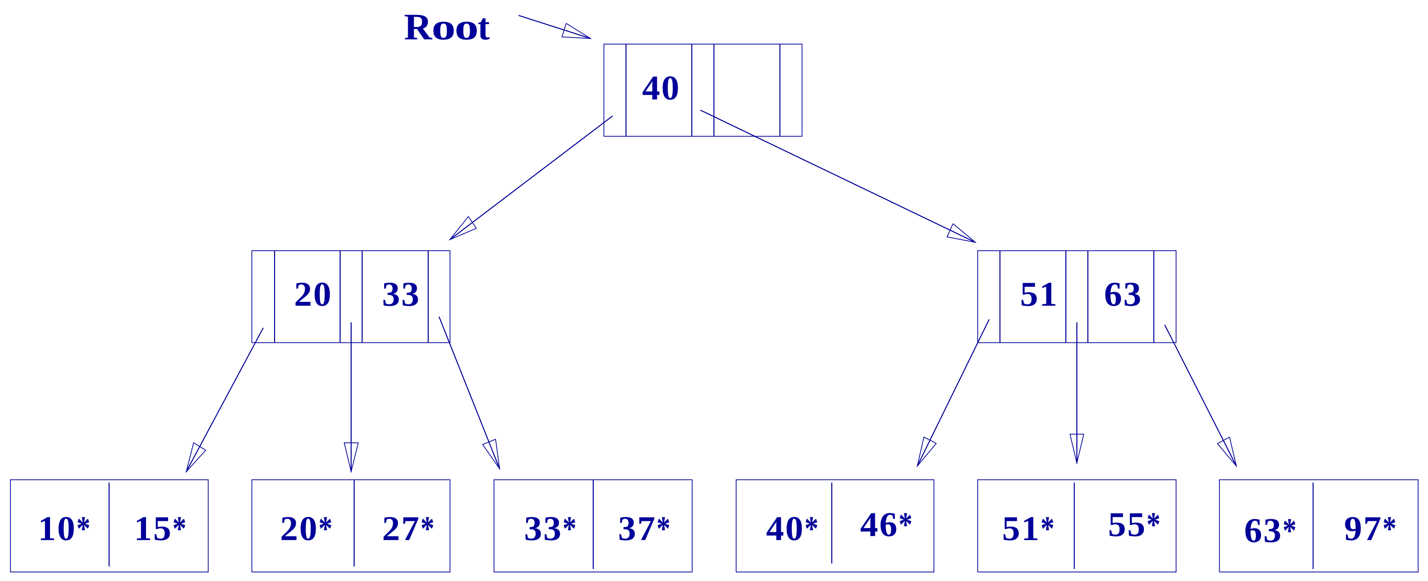
- Create file(s): Data entries are sorted. Leaf data pages are allocated sequentially. Index is constructed. Space for overflow pages is allocated.
- Find an entry (search). Obvious generalisation of method for binary search tree.
  - If pages are large, we can also do binary search on a page, but this may not be worth the effort. I/o costs dominate!
- Insert an item. Find leaf data page (search) and put it there. Create overflow page if needed.
- Delete and item. Find leaf data page (search) and remove it. Maybe discard overflow page.

Note. In ISAM, the index remains *fixed* after creation. It is easy to construct pathological examples which make ISAM behave badly.

A simple ISAM example

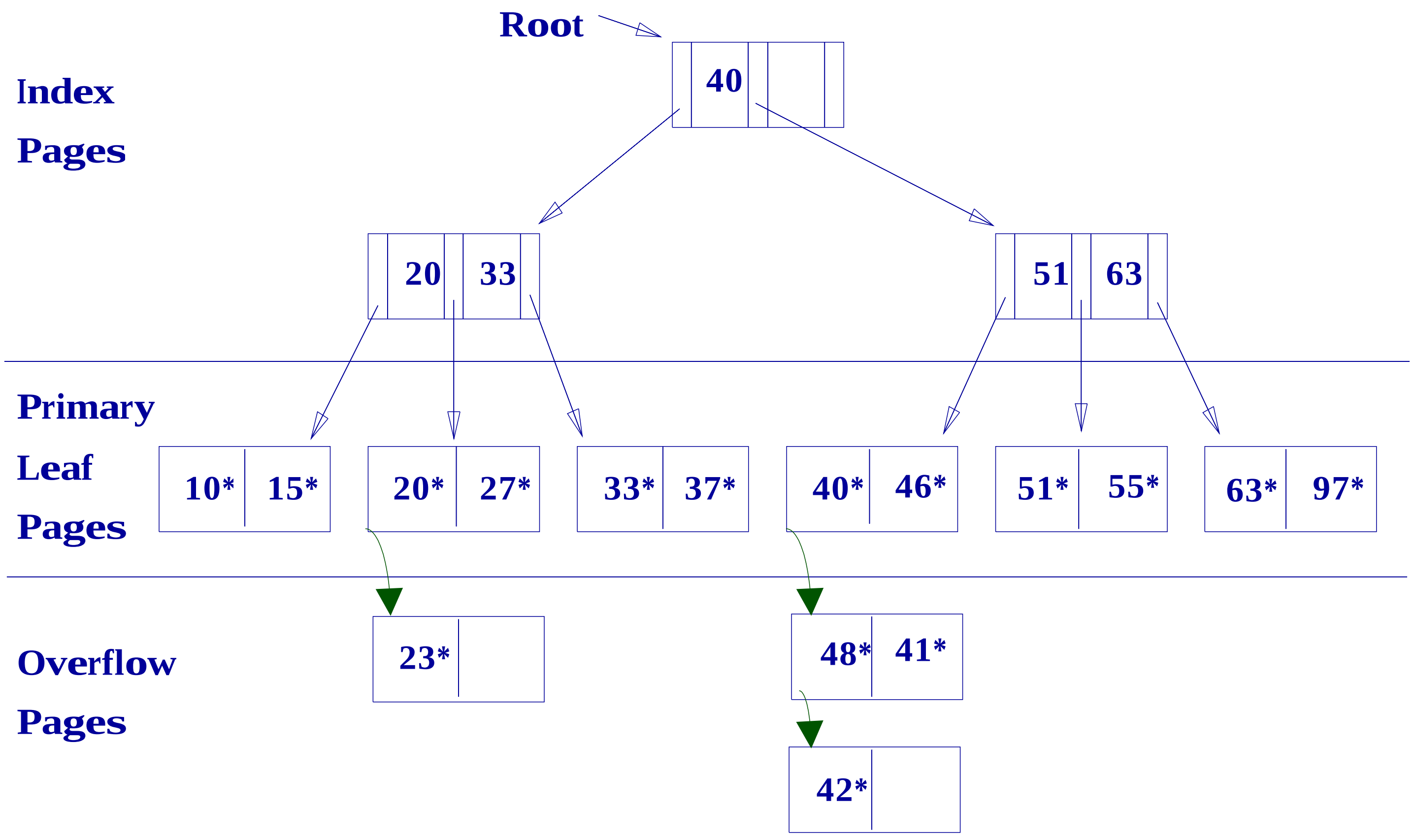
This is not realistic. The example is only a 3-ary tree. In practice one might have 100-way branching.

Note that we can perform an ordered traversal of the data entries by a traversal of the index.



ISAM after inserts

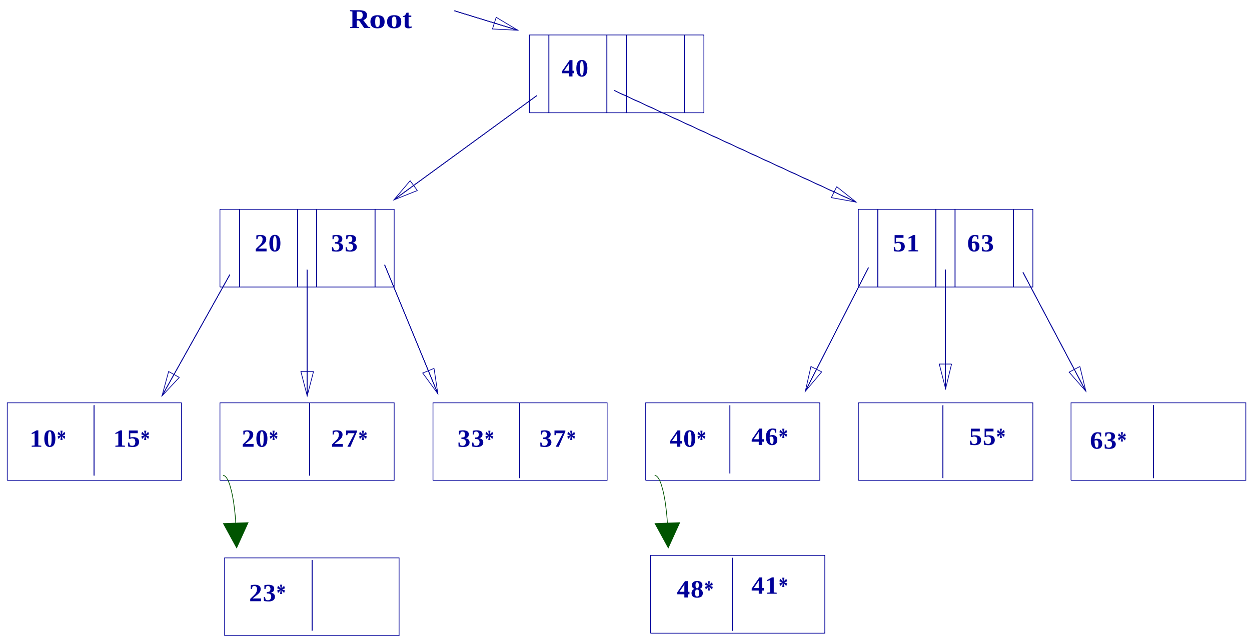
Inserting $23^*, 48^*, 41^*, 42^*$



ISAM after deletes

Deleting $42^*, 51^*$

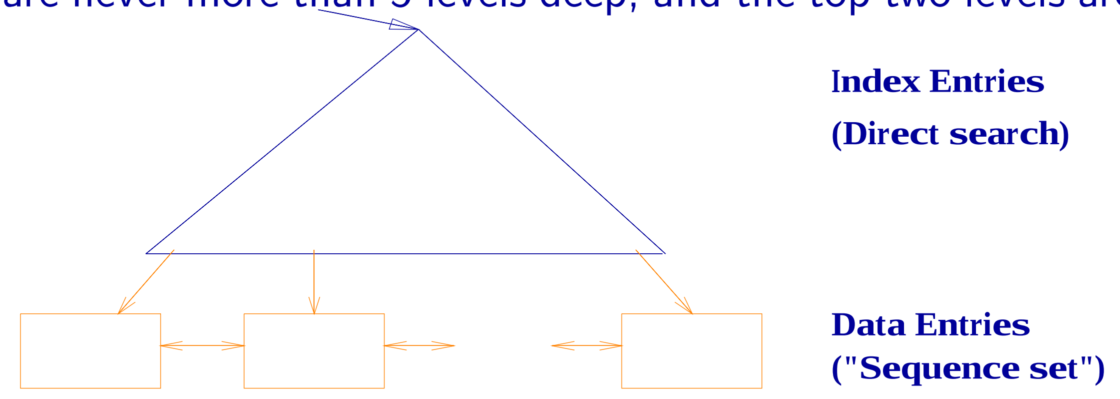
Note that 51 appears as a key but no longer as a leaf value.



Main problem with ISAM: *index is fixed*. Can become worse than useless after a long series of inserts and deletes.

B+ Tree. The Standard Index

- Each node (page) other than the root contains between d and $2d$ entries. d is called the *order* of the tree. Pages are not always full.
- Suitable for both equality and range searches.
- Lookup (equality), insertion and deletion all take approx. $\log_k(N)$ page accesses where $d \leq k \leq 2d$.
- Tree remains perfectly balanced! All the leaf nodes are the same distance from the root.
- In practice B-trees are never more than 5 levels deep, and the top two levels are typically cached.

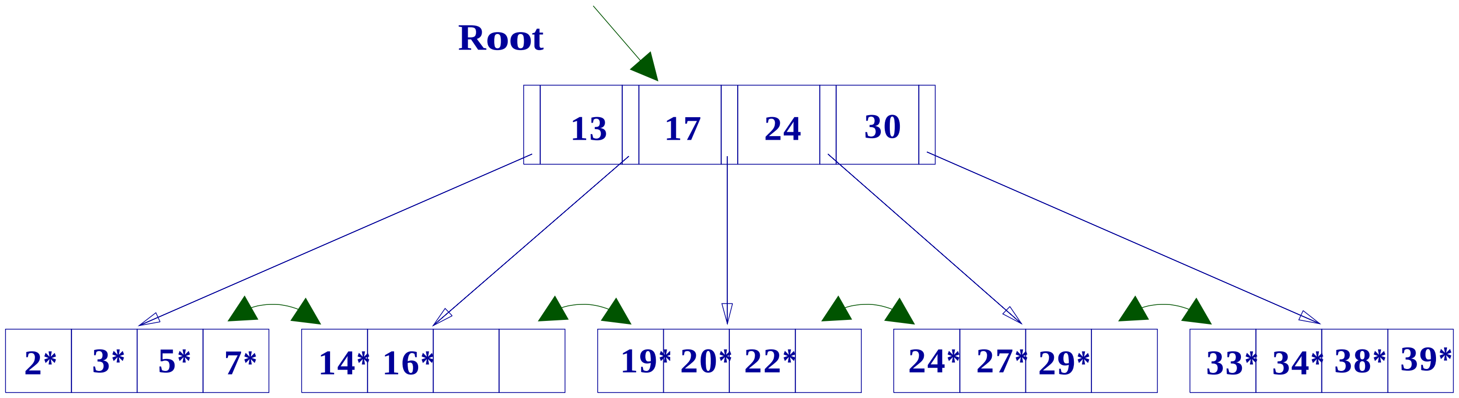


Example B+ Tree

Search is again the obvious generalization of binary tree search. Could do binary search on nodes.

Key values need not appear in any data entries.

Data pages are linked (we'll see why shortly)



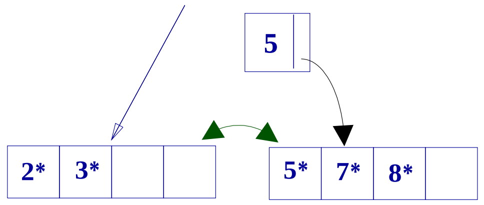
Inserting into a B+ Tree

- Find leaf page that should hold k^*
- If page is not full, we can insert k^* and we are done.
- If not, split the leaf page into two, putting "half" he entries on each page and leaving a middle key $k'$to be inserted in the node above.
- Recursively, try to insert k' in the parent node.
- Continue splitting until *either* we find a node with space, *or* we split the root. The root need only contain two children.

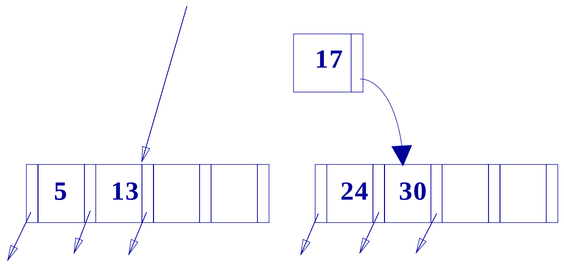
Either the tree grows fatter or (very occasionally) it grows deeper by splitting the root.

Example Insertion

Inserting 8^*. The page is full so the leaf page splits and 5, the middle key, is pushed up. (5^* remains in the leaf page.)

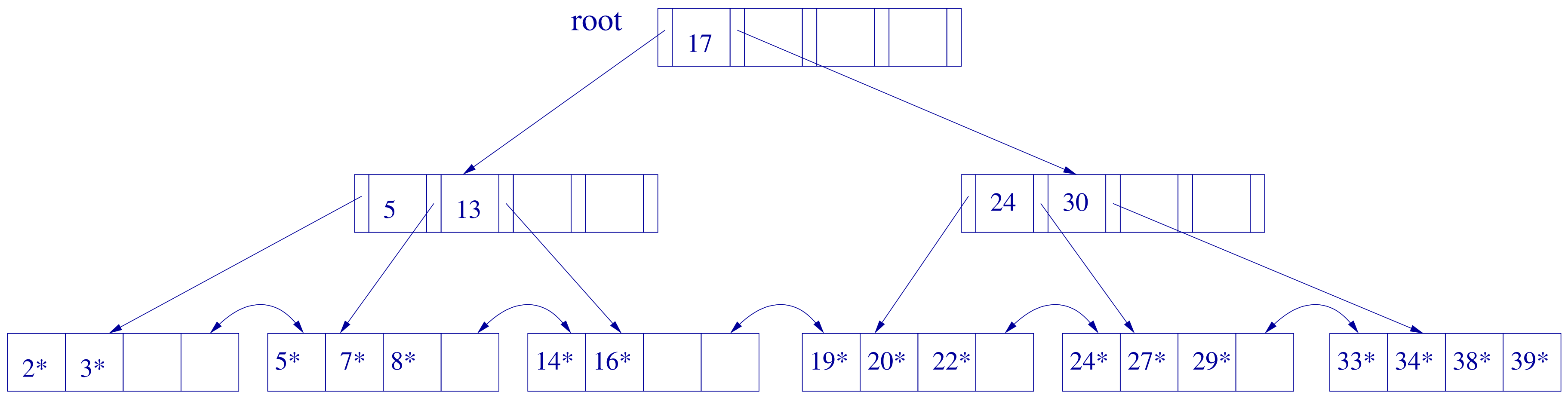


The parent page is also full so this page splits and 17 is pushed up. (This is not in a data entry, so it does not remain on one of the child pages.)



The new root contains just 17

The Resulting Tree



We could have avoided the root split by sideways redistribution of data on the leaf pages. But how far sideways does one look?

Deleting from a B+ Tree

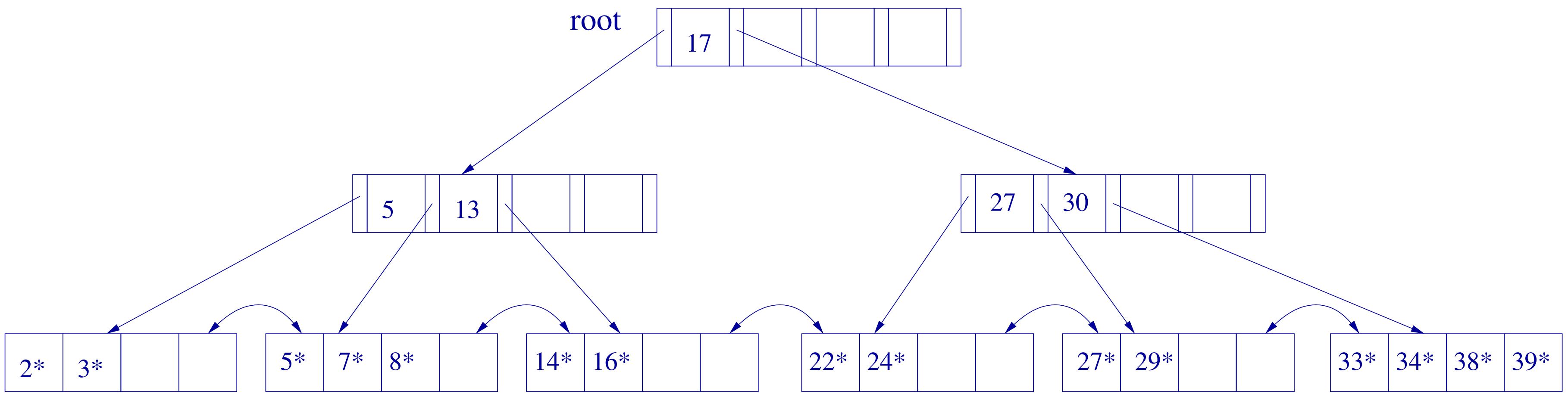
- Find leaf page with entry to be deleted.
- Remove entry. If page is still half full we are done.
- Otherwise look at adjacent siblings with the same parent. Is there one that is more than half full? *Re-distribute* (involves parent) and we are done.
- If not, *merge* with an adjacent sibling with same parent.
- Key in parent node must now be removed.
- If needed, recursively merge – possibly removing the existing root.

After Some Deletions

Having inserted 8^* we delete 19^* and then 20^*

Deleting 19^* does not cause the page to become less than half full.

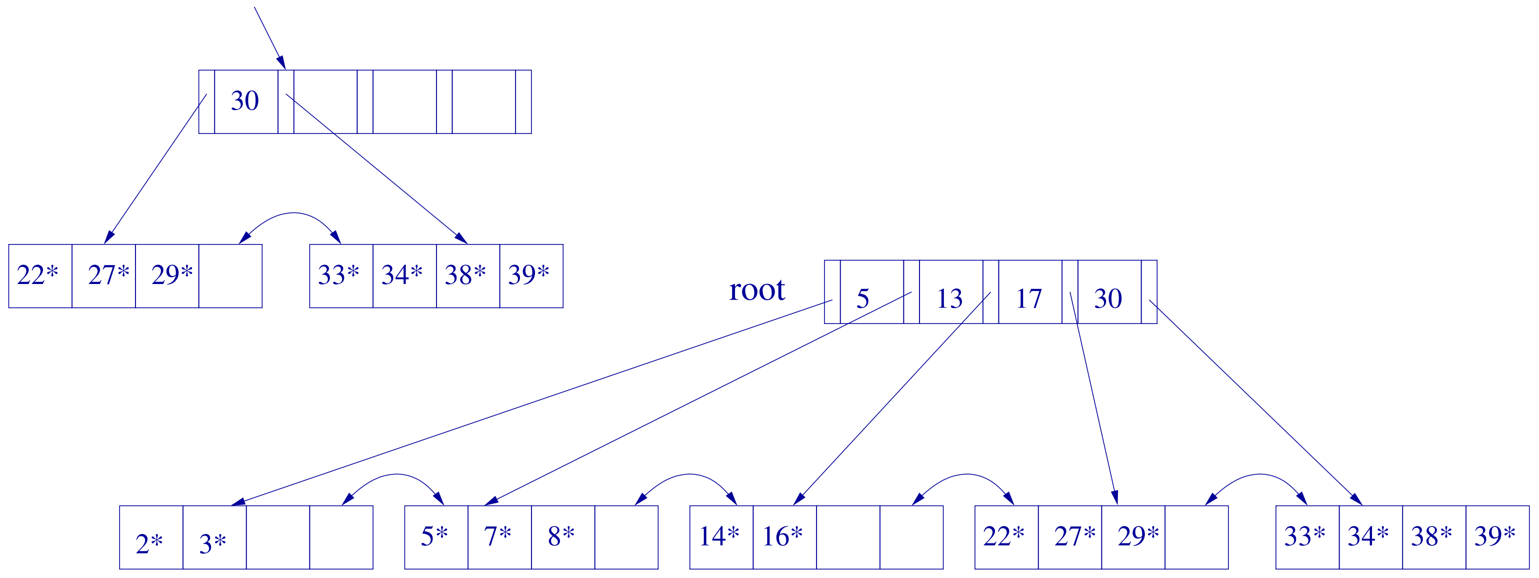
Deleting 20^* can be done by moving 24^* from the adjacent page and moving a new key, 27, up.



One more deletion...

We delete 24^*. Need to merge with adjacent sibling and remove key 27 from parent.

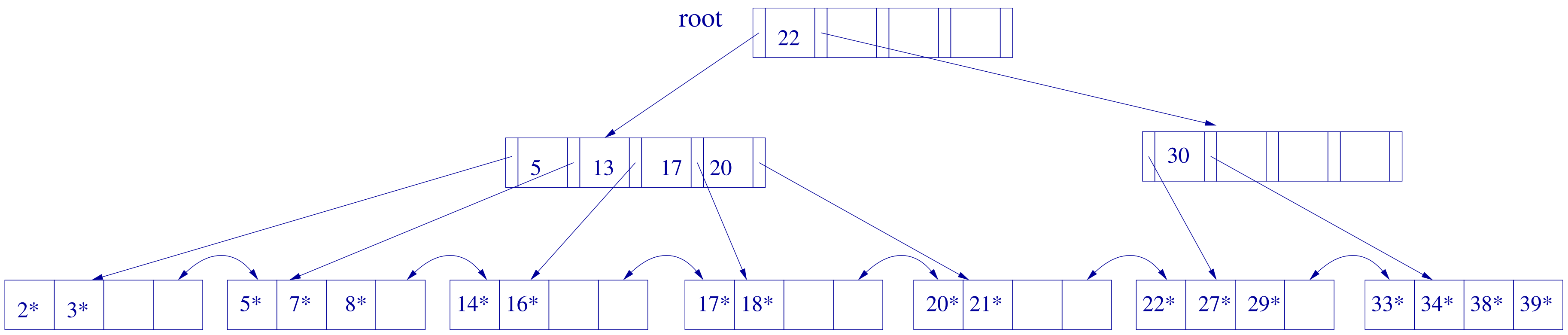
Parent is now too empty and must be merged with left sibling.



Redistributing at Internal Nodes

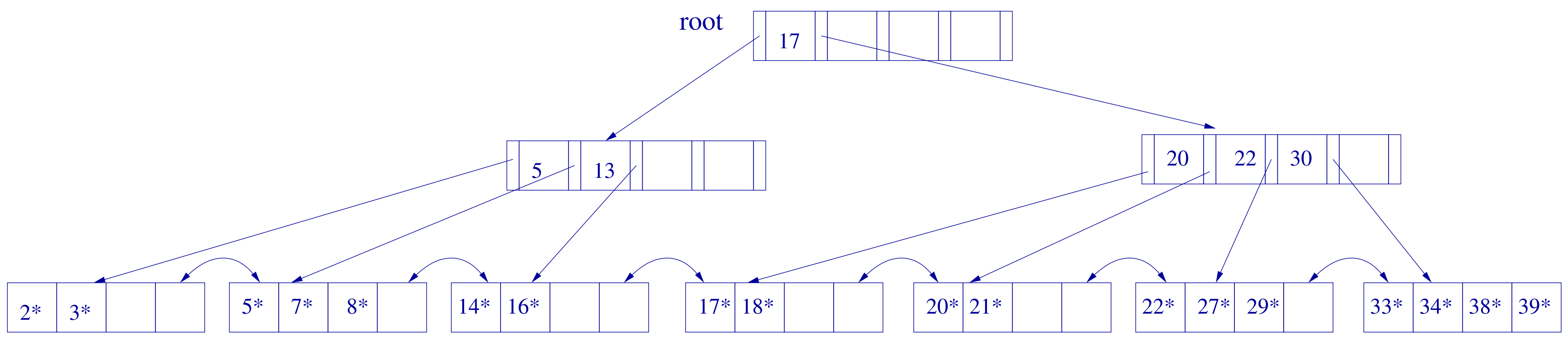
Here we have just deleted 24 from some other tree.

Current tree is "unfinished". We cannot merge node containing 30 with adjacent node, but we can redistribute



After Redistribution

We could have moved just 20; we've also moved 17



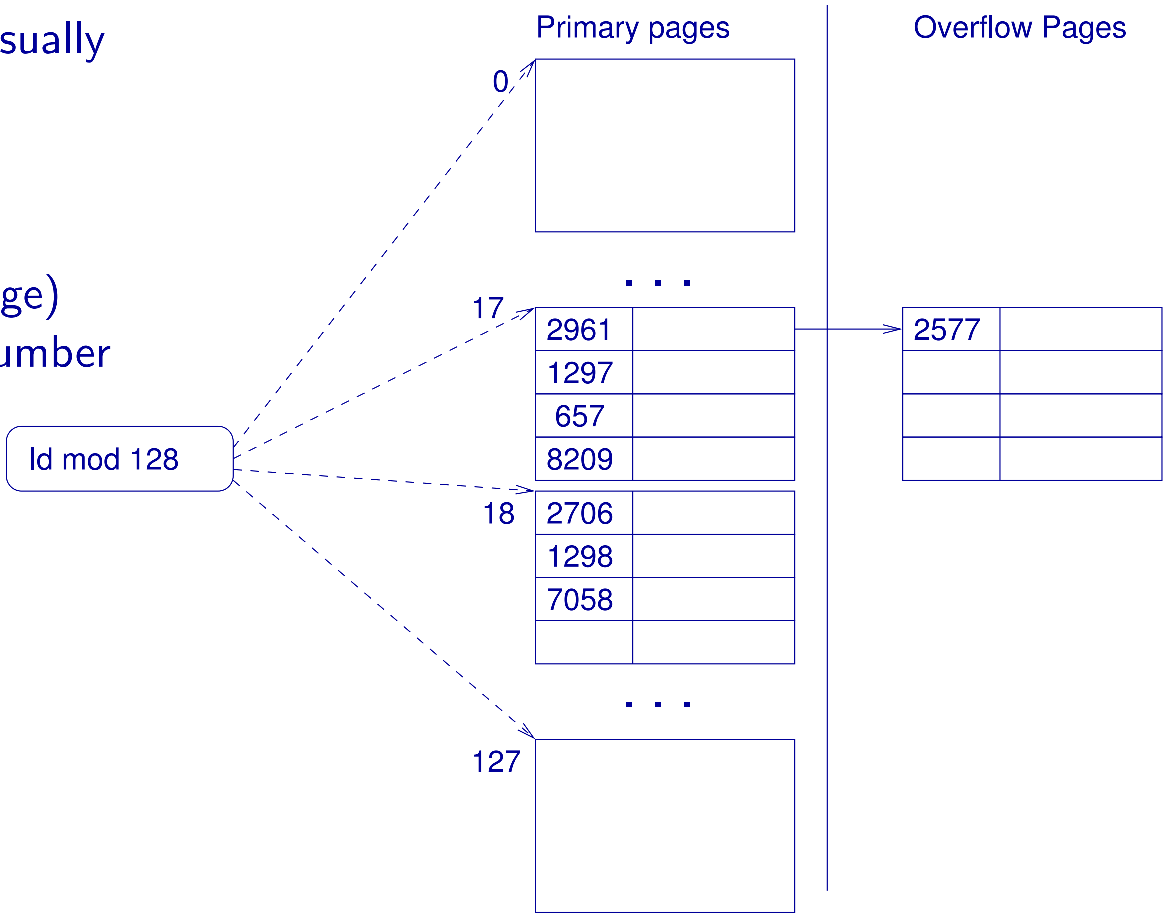
B+ Trees in practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 1334 = 312,900,700 records
  - Height 3: 1333 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Hashing

An alternative to tree indexing

- Lookup, insert and update usually take 1 read *except when restructuring is needed.*
- No good for range searches.
- Typically hash to bucket (page) $h(x)$ mod m where m is number of pages.



Expanding Hash Tables

If the hash table becomes "full", which can happen because

- we don't have overlow pages and a page becomes full, or
- the number (ratio) of overflow pages exceeds some threshold,

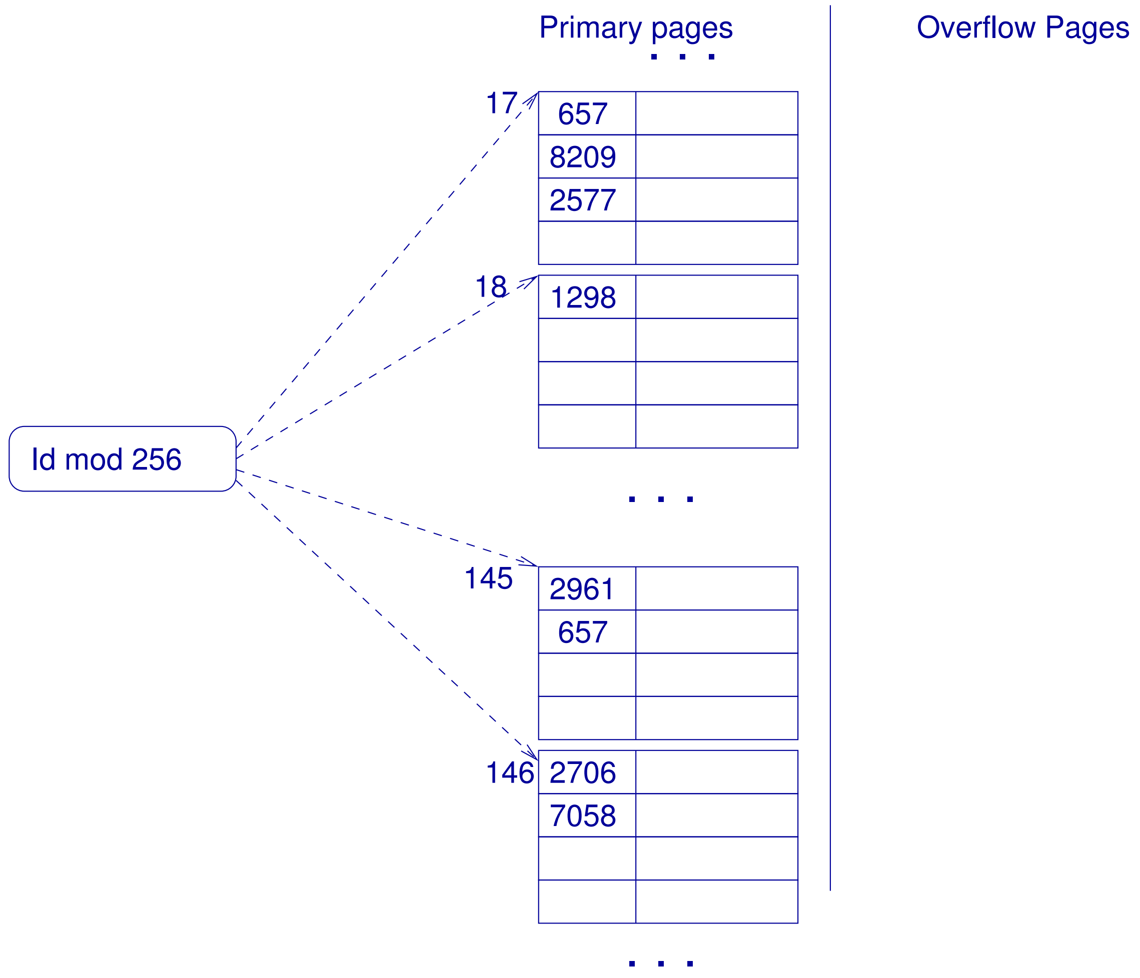
we need to *restructure* the hash table.

We can double the number of buckets and split each one.

E.g. $1297 \bmod 128 = 17 = 1297 \bmod 256$. So 1297 stays on page 17.

However $2961 \bmod 128 = 17$ but $2961 \bmod 256 = 145$, so 2961 gets moved from page 17 to page 145.

Doubling the table size

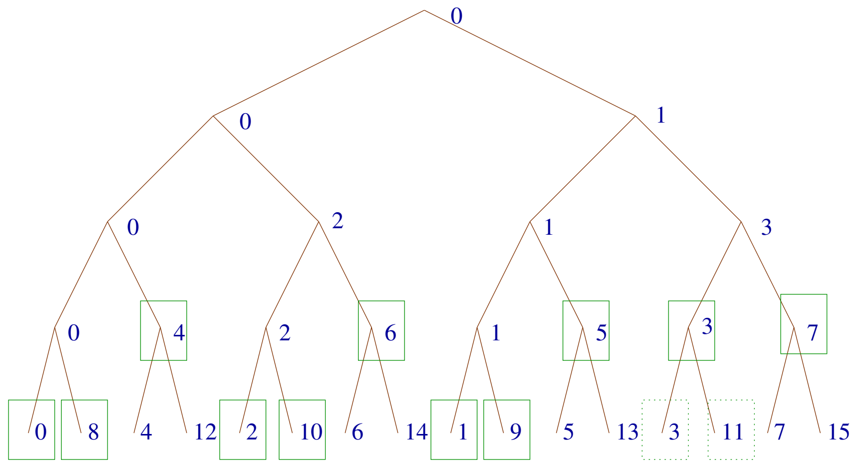


Alternative Hashing Techniques

Reconfiguring a whole hash table because can be very time consuming. Need methods of "amortizing" the work.

- *Extendible hashing* Keep an index vector of n pointers. Double the index when needed, but only split pages when they become full.
- *Linear Hashing*. Clever hash function that splits pages one at a time. This does not avoid overflow pages, but we add a new page when the number of records reaches a certain threshold. (80-90%) of maximum capacity.

Linear Hashing



11 buckets with bucket 3 about to split

Suppose we want k buckets
Choose n s.t. $2^n \leq k \leq 2^{n+1}$
Assume we have h – a hash function into some large range.

This diagram shows how the buckets split. Rearrange in order to see their layout in memory. N.B. New bucket is always appended.

$$\begin{aligned} \text{Bucket for } x \quad &= \quad h(x) \text{ mod } 2^n \qquad && \text{if} \quad (h(x) \text{ mod } 2^n) \geq 2^n - k \\ &= \quad h(x) \text{ mod } 2^{n+1} && \text{otherwise} \end{aligned}$$

Bucket to split when we increment k to $k+1$ is bucket $k - 2^n$ All other buckets stay put.

Review

- Properties of storage media
- Caching
- Placemant of tuples
- File Organization
- Relative merits of heap files, hashed files, ordered files.
- What indexes provide
- Clustered/unclustered indexes
- ISAM
- B-trees: lookup, insertion and deletion.
- Hash tables – basic properties.

Implementing Relational Operations

Reading: R&G 12.

The figures and example are taken from this chapter.

We'll consider implementation of the following operations:

- Join ($\bowtie$)
- Selection (σ)
- Projection (π)
- Union ($\cup$)
- Difference ($\setminus$)
- Aggregation (`GROUP-BY` queries)

Union and difference are closely related to *join*, which is by far the most interesting operation.

Object-oriented databases – a brief digression

The idea is that an object-oriented programming language (C++, Java, etc.) with minor embelishments gives us a DBMS.

Example. Objectstore is an extension of C++ with "persistent" objects in secondary storage. In Java the "DDL" would look something like:

```
class Department {
  extent Departments
  int DeptId
  char DName[20]
  string Address
  ...}
```

```
class Employee {
  extent Employees
  int EmpId
  char Name[30]
  Department Dept // a "pointer"
  ...}
```

`Employees` and `Departments` are global names (like table names) and contain "all" the `Employee` and `Department` objects.

OQL, an object-oriented QL

OQL:

```
SELECT   e.Name, e.Dept.DName
FROM     Employees e
WHERE    e.Age > 40
```

SQL:

```
SELECT   e.Name, d.DName
FROM     Employees e, Departments d
WHERE    e.Age > 40
AND      e.DeptId = d.DeptId
```

Which is more efficient?

Sometimes it is better to do a join than a set of derefences. It may pay to “optimize” the OQL query to an SQL query!!

An example for relational operator optimization.

- Table R: p_R tuples/page, M pages $(= Mp_R)$ tuples.
- Table S: p_S tuples/page, N pages $(= Mp_S)$ tuples

Some plausible numerical values. When pages are 4000 bytes long, R tuples are 40 bytes, and S tuples are 50 bytes long.

tuples/page	# pages	# tuples
$p_R = 100$	$M = 1000$	$100,000$
$p_S = 80$	$N = 500$	$40,000$

Equality join on one column

```
SELECT   *
FROM     $R$, $S$
WHERE    $R.i = S.i$
```

Assume i is a key for S and is a foreign key in R, thus Np_S tuples in output

Simple nested loop join:

```
for each tuple $r \in R$
    for each tuple $s \in S$
        if $r_i = s_i$ then add $r, s$ to output
```

Cost (of I/O) $= M + p_R M N = 1000 + 100 \times 1000 \times 500 = 50,001,000$. This is the cost of reading all the pages of R plus the cost of reading all the pages of S for each *tuple* in R.

Page-oriented Nested Loop Join

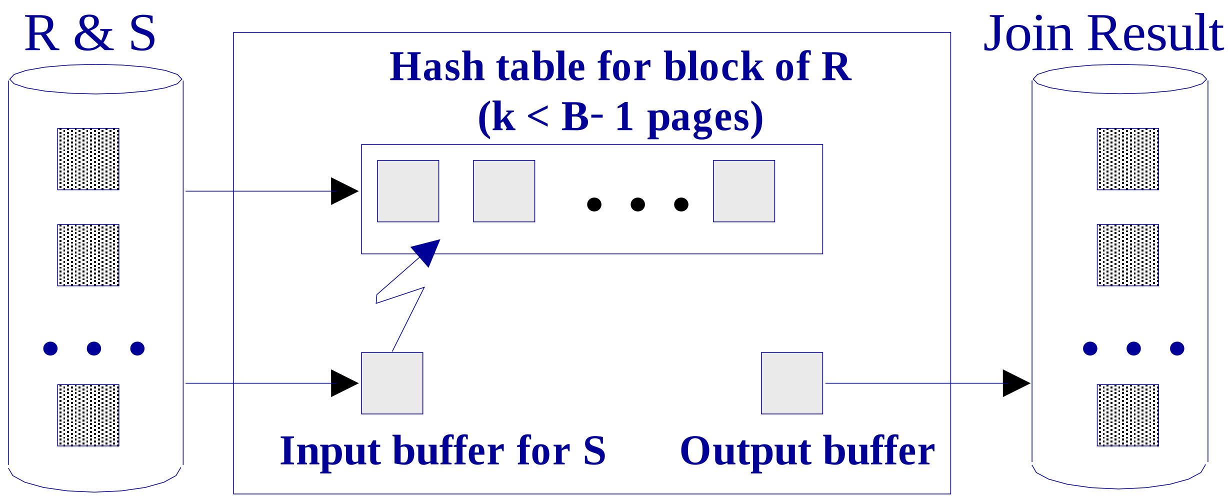
```
for each page $B_R$ of $R$
  for each page $B_S$ of $S$
    for each tuple $r \in B_R$
      for each tuple $s \in B_S$
        if $r_i = s_i$ then add $r, s$ to output
```

Cost is now $M + MN = 1000 + 1000 \times 500 = 501,000$ (Read all pages of R + all pages of S for every page of R

If we interchange the "inner" and "outer" tables, we get $N + NM = 500 + 1000 \times 500 = 500,500$

Block Nested Loop Join

Extension of page-oriented nested loop join. We read as many pages of the outer table into the buffer pool as we can (a "block" of pages).



Note that we can make each block a hash table to speed up finding matching tuples.

I/O cost of block nested loop join

Cost: Scan of outer table $+$ # outer blocks $\times$ scan of inner table

(# outer blocks $=$ $\lceil$# pages of outer / blocksize$\rceil$)

- With R as outer, and blocksize$=100$:
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, we scan S; 10×500 I/Os.
  - TOTAL: $6,000$ I/Os
- S as outer, and blocksize$=100$:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan R; $5 * 1000$ I/Os.
  - TOTAL: $5,500$ I/Os

Index joins

```
for each tuple $r \in R$
    for each tuple $s$ with key $r_i$ in index for $S$
        add $r, s$ to output
```

Suppose average cost of lookup in index for S is L.

Cost of join is # of pages in R plus one lookup for each tuple in R. $M + MLp_R = 1000 + 100,000L$

If $L = 3$, this is $301,000$

The *minimum* value for L is 1 (when tuples of S are stored directly in hash-table buckets.) Total is then $101,000$

Sort-merge join

- Sort both R and S on join column
- "Merge" sorted tables

Cost of sorting. Under reasonable assumptions about the size of a buffer pool, external memory sorting on 1000 pages can be done in two passes. Each pass requires us to read and write the file.

Note. Sorting $m = 100,000$ tuples requires $m \log_2 m \approx 1,700,000$ main memory comparisons. The I/O time in this example dominates.

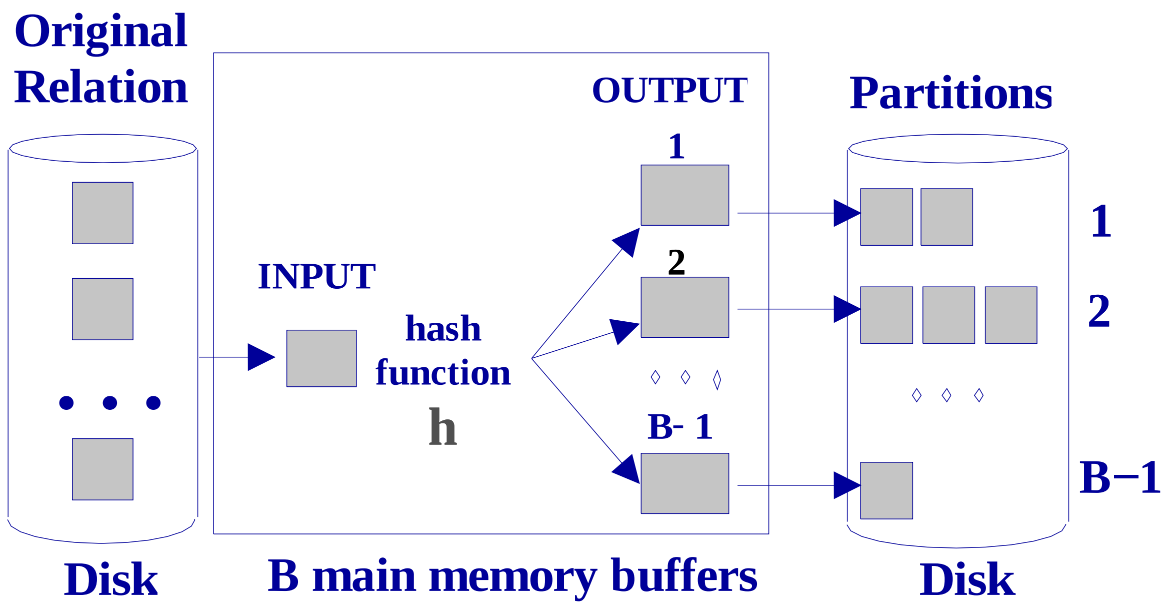
Cost of sorting R and S is $4 \times 1,000 + 4 \times 500 = 6,000$

Cost of Merge $= 1,500$

Total: $7,500$

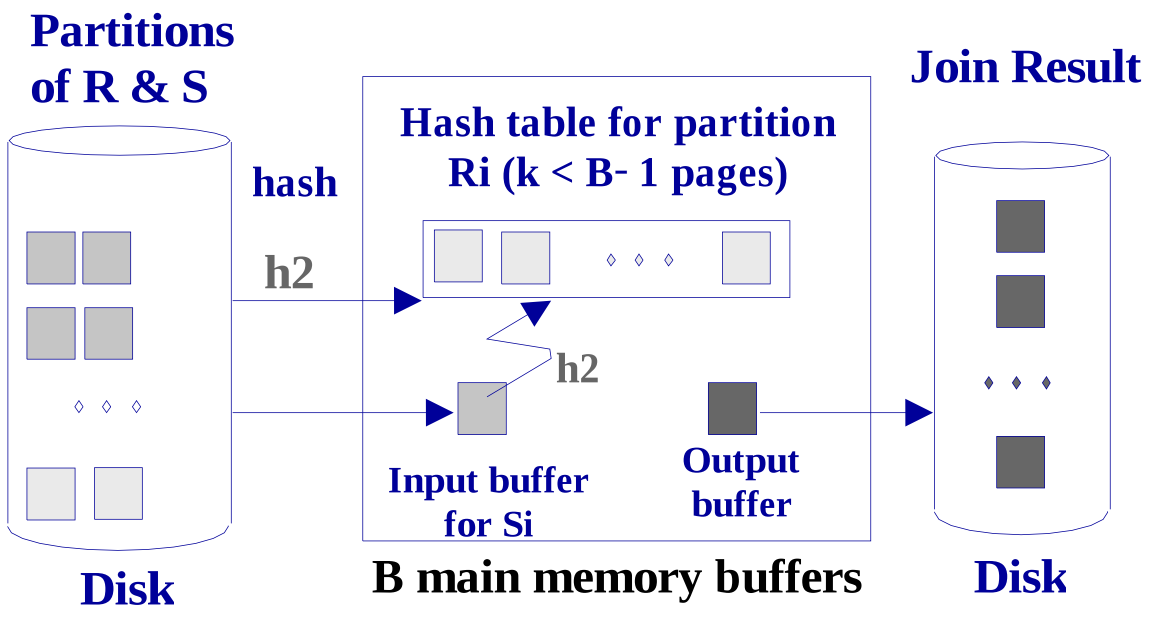
Hash join – phase 1

Partition both relations using hash function h applied to the join column values: R tuples in partition i will only match S tuples in partition i.



Hash join – phase 2

We do a block-nested join on each pair of partitions: read in a partition of R, hash it using h_2 (*must be different from* h) then scan matching partition of S and search for matches.



Observations on Hash-Join

- The number of partitions is determined by the number of buffer pages, $B - 1$
- We want each partition (of R) to fit into the buffer memory, the partitions should have less than $B - 2$ pages.
- Thus each table should occupy less than $\approx B^2$ pages.
- Can apply the technique recursively if needed.

Cost of hash-join $=$ 3 passes through both tables, $3(M + N) = 4,500$!!

Hash-join is parallelisable.

General Join Conditions

- Equi-join on several fields. Straightforward generalisation of join on one field. Can make use of index on all or any of the join columns.
- Inequality joins, e.g., $R \bowtie_{R.i<S.i} S$. Hash joins inapplicable. Variations on sort-merge may work. E.g. for "approximate" joins such as
  $R \bowtie_{S.i-C<R.i \ \wedge \ R.i<S.i+C} S$.
  Indexing may work for clustered tree index.
  Block nested loop join is a good bet.

Selection

Single column selections, e.g. $\sigma_{i=C}(R)$ and $\sigma_{i>C}(R)$.

- No index on i – scan whole table.
- Index on i
  - Hash and tree index OK for equality.
  - Tree index may be useful for $\sigma_{i>C}(R)$, especially if index is clustered.
    If tree index is unclustered:
    - Retrieve qualifying rid's.
    - Sort these rid's.
    - Retrieve from data pages with a "merge".

Selection on “complex” predicates

General problem, can indexes do better than a complete scan of the table. Look for special cases.

- Range queries, $\sigma_{A<i \wedge i<B}(R)$. Use tree index.
- Conjunction of conditions, e.g. $\sigma_{i=A \wedge j=B}(R)$
  - Index on (i, j) – Good!
  - Index on i. Obtain tuples and then perform $\sigma_{j=B}$.
  - Separate indexes on i and j. Obtain and sort rid’s from each index. Intersect (use merge) the sets of rid’s.

Projection

Only interesting case is a "real" projection, `SELECT DISTINCT` $R.i$, $R.j$ `FROM` R.

- Eliminate duplicates by sorting
  - Eliminate unwanted fields at first pass of sort.

If the projection fields are entirely contained within an index, e.g., we have an index on (i, j), we can obtain results from index only.

Other set operations

- Intersection is a special case of join.
- Union and difference are similar. Again, the problem is eliminating duplicates.
  - Sorting based approach:
    - Sort both relations (on combination of all attributes).
    - Scan sorted relations and merge them.
  - Hash based approach to:
    - Partition R and S using hash function h.
    - For each S-partition, build in-memory hash table (using $h2$), scan corresponding R-partition and add tuples to table while discarding duplicates.

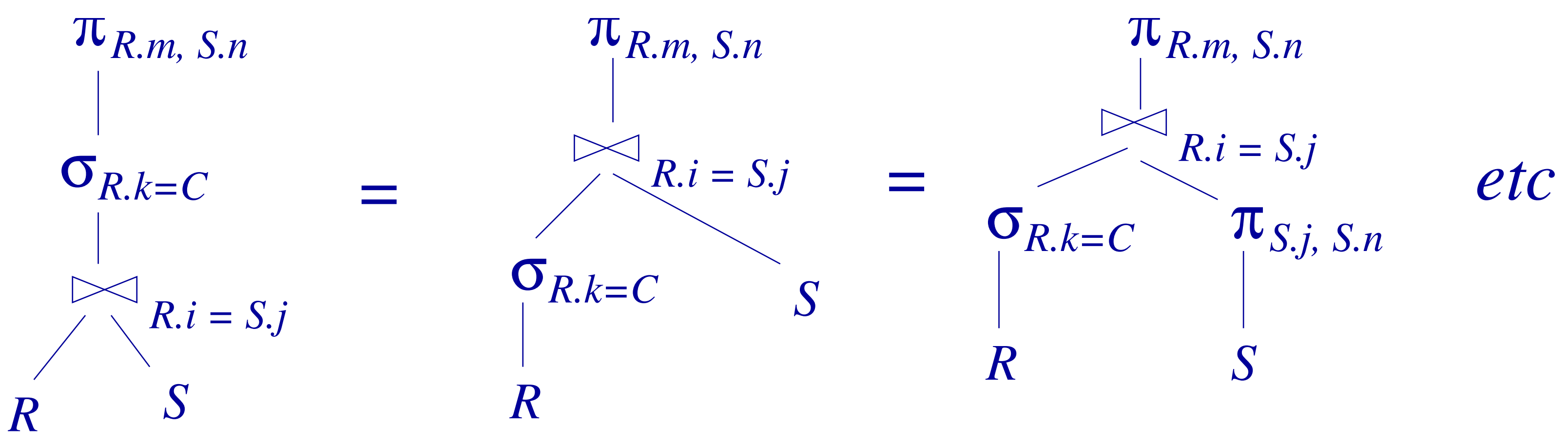
Query optimization – brief notes

We have examined how to evaluate each relational operation efficiently. How do we evaluate a whole query?

We make use of the rich set of *rewriting rules* for relational algebra. Examples:

- *Join re-ordering*. $R \bowtie S \bowtie T = R \bowtie S \bowtie T$. Do the most “selective” join first.
- *Pushing selection through ....* $\sigma_{R.i=C}(R \bowtie S) = \sigma_{R.i=C}(R) \bowtie S$.

Problem: the search space of expressions can be very large. In practice query optimizers explore a small subset of the possibilities.



Cost-based optimization

Whether or not a particular rewriting is a good idea depends on the statistics of the data. Consider pushing selection through joins. Is it a good idea in the following queries?

```
SELECT   E.Name, D.DName
FROM     Employee E, Department D
WHERE    E. DeptId = D.DeptId
AND      D.Address = "KB"
AND      E. Age < 20
```

```
SELECT   E.Name, D.DName
FROM     Employee E, Department D
WHERE    E. DeptId = D.DeptId
AND      D.Address = "KB"
AND      E. Age < 60
```

Keeping statistics of tables such as the *selectivity* of an attribute is needed to decide when a rewriting is useful. This compounds the problem of finding the optimum.

Implementing Relational Operations - Review

- Joins
  - Page-oriented joins
  - Block-nested loop joins
  - Index-based joins
  - Sort-merge joins
  - Hash joins
- Using indexes in selections.
- Projection – elimination of duplicates
- Union and Difference
- Query rewriting, why algebraic identities are useful
- Cost-based optimization.