

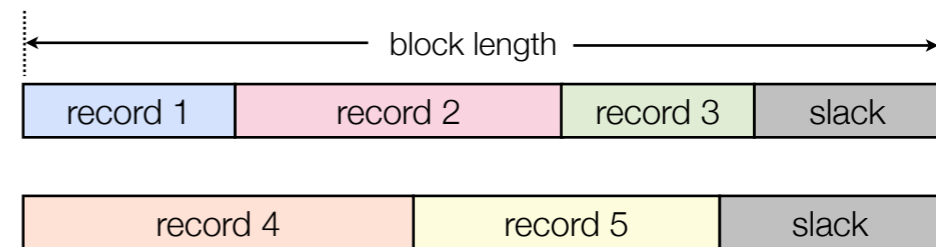
COMP2400

Relational Databases

Lecture 27: File Layout and Indexing

Ben Lippmeier
Australian National University
Semester 2
2008

Block allocation and slack space

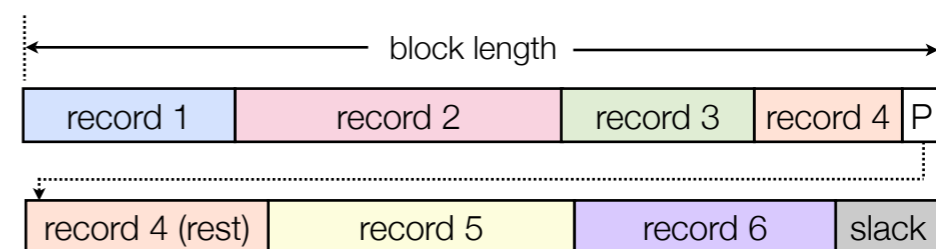


- When variable length records are stored in blocks on the disk, some space may be left over at the end of each block.
- The first *slack space* is not large enough to contain record 4.
- For databases large with fixed size records, designing the records to fit within disk blocks can increase the speed and space efficiency of storage.
- Block lengths of 1k, 4k, 8k and 16k bytes are typical.

The next three weeks:

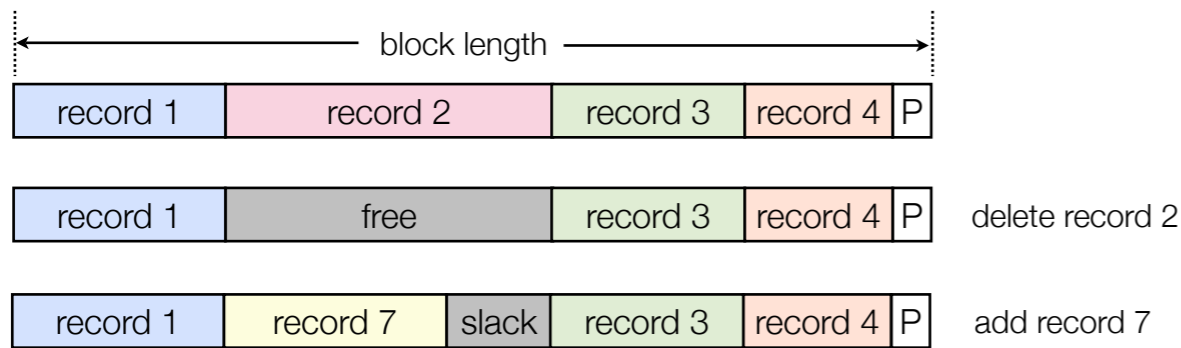
- One lecture each on Wednesday and Thursday through week 13.
- Topics covered are
 - File Layout and Indexing
 - Binary and B-Trees for Indexing
 - Database Hardware inc RAID
 - Query Optimisation (3 lectures)
- If you have any questions on this material
email: Ben.Lippmeier@anu.edu.au
office: N312, top floor of Comp Sci building
- Midsemester exam viewings on Monday 20th, 3-5pm.

Spanned records reduce slack space



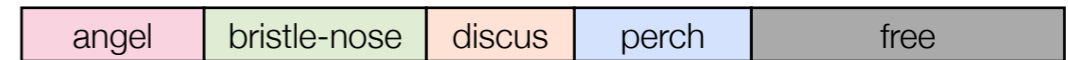
- Slack space can be recovered by spanning records over multiple blocks. The front of a spanned record contains a pointer to the block holding the rest.
- We have eliminated some slack space, at the cost of requiring two block accesses every time we read or write record 4.
- Hybrid spanned / unspanned systems are possible, where a record is only split if the slack space created would be above a certain threshold.

Space fragmentation



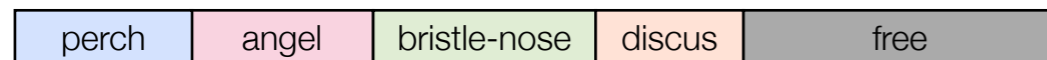
- When a record is deleted, we end up with free space in the middle of a block.
- Allocating a new record of a different size into the free space can create slack which is too small to contain a new record.
- One solution is to keep a table of free space in each block, and try and allocate a new record into a section of free space of similar size.

Ordered files enable binary search



- If we physically order records on their key fields, we can find the record with a specific key via binary search.
- Naive linear search accesses $r/2$ records on average, where r is the total number of records in the table.
- Binary search accesses $\log_2(r)$ records on average.
eg $\log_2(16) = 4$, $\log_2(1k) = 10$, $\log_2(1M) = 20$

Heap files are unordered



- Records in *heap files* appear in the order they are inserted in.
- Inserting a new record into a heap is very fast.

Just keep a pointer to the start of the free space. To insert, write the record into the free space and add the length of the record to the pointer.

- Finding a particular record can be very inefficient.

Without any additional information telling us where to look, we must examine every record in the table to find the one we're interested in.

Block headers



header:

slot	offset	species
0	128	angel
1	240	bristle-nose
2	480	discus
3	638	perch
4	762	free

- For binary search, we usually need some sort of *header* for each block to record where each record starts. Headers can be per-block, file or table.
- Headers can be stored with their associated blocks/tables, or separately. Caching headers separately from the data allows us to find the record with a particular key, without reading irrelevant data from the disk.

Inserting into ordered files



slot	offset	species
0	0	angel
1	240	bristle-nose
2	762	cichlid
3	480	discus
4	638	perch
5	948	free

↓
copy entries to
to later slots in
header

- When inserting a record into the middle of an ordered table, on average we must copy half the entries to later slots to maintain the ordering.
- For blocks with headers, we can keep the header ordered but leave the records in the block unordered. Records are usually larger.

Hashing key values to get their slot numbers.

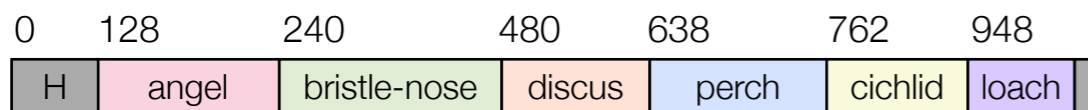
- For string keys, a common hash function is to sum the ASCII values of each character, divide the sum by the highest slot index in the table, and take the remainder: ie, in Haskell:

```
sum (map ord "perch") `mod` 8 => 2
```

- A good hash function produces a random distribution of output values.
- To find a record with a particular key, simply hash the key and lookup its offset from the file header.

slot	offset	species
0	186	tetra
1
2	240	perch
3	440	discus
4	380	cichlid
5
6
7	68	bristle-nose

Overflow bins for fast insert into ordered files.



header:

slot	offset	species
0	128	angel
1	240	bristle-nose
2	480	discus
3	638	perch

overflow:

slot	offset	species
0	762	cichlid
1	948	loach
2	1002	free
3

- The cost of insert can be mitigated by providing an additional, unordered overflow bin in the header.
- New records are added to the overflow bin which is periodically merged with the main header.

Hash collisions

- When two keys have the same hash they are said to *collide*. If there is already a key in the header with this hash then we must store the offset for that record in a different slot.

```
sum (map ord "tuna") `mod` 8 => 0
```

- One method for *collision resolution* is to store the slot number for subsequent keys in an extra column in the header.
- Here, the keys “tetra”, “oscar” and “tuna” all hash to slot 0.

slot	offset	species	chain
0	186	tetra	1
1	758	oscar	5
2	240	perch	...
3	440	discus	...
4	380	cichlid	...
5	950	tuna	...
6
7	68	bristle-nose	

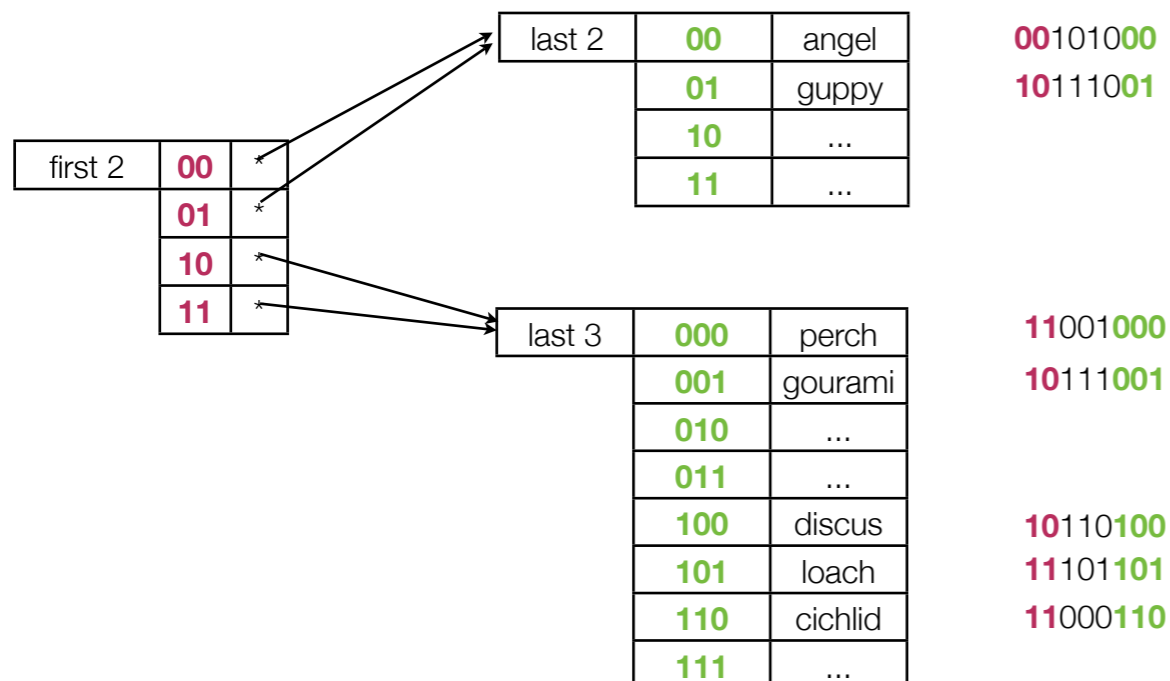
Extendible Hashing

- A drawback of *static hashing* is that it is hard to increase the size of a table.
- We must discard the old table and re-insert all the record into a larger one.
- Instead, we can treat the binary hash value as an index into a tree.
- When a node is too large or has too many collisions then locally reorganise the tree. Local reorganisations do not affect unrelated data.
- There are many possible options for tree structure and reorganisation heuristics.
- Choose one that suits the underlying hardware.

Indexing

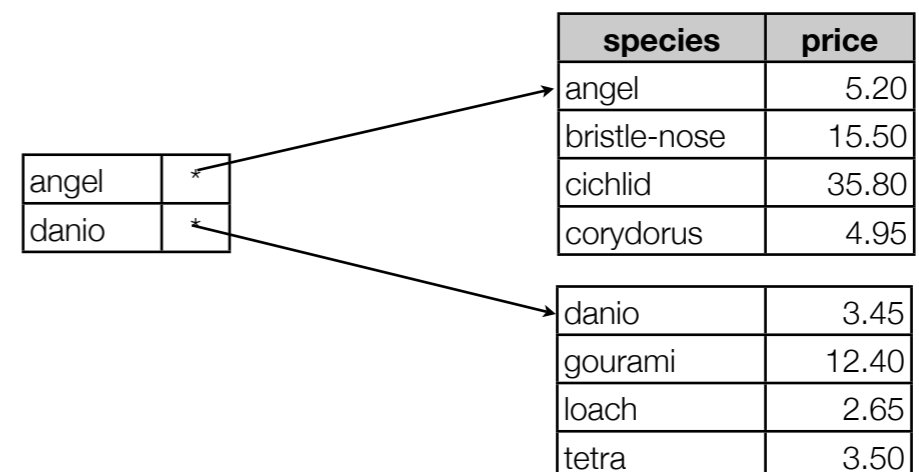
- Indexes provide *access paths* to efficiently find records in response to certain search conditions.
- There can be multiple indexes per table. Suitable indexes to use depend on the properties of the data and likely search queries.
- The headers we have seen are *primary indexes*, as they are used to physically order the records on the disk.

Extendible Hashing cont...



Dense vs Sparse indexes

- A dense index has an entry for *every record in the table*. Sparse index only have entries for the first record in each block.
- The *blocking factor (bfr)* is the number of records per block. We only need $(total\ records) / (bfr)$ entries in a simple sparse index.



Clustered Indexing

- Clustered indexes are used for search fields which may have repeated values.
- A typical DBMS will reserve a whole disk block to hold the record pointers for each search value.

index for species_origin(origin)

origin	index
Australia	0
Asia	1,2,6
South America	3,5
Africa	4

species_origin

slot	species	origin
0	silver perch	Australia
1	bristle-nose	Asia
2	gourami	Asia
3	discus	South America
4	cichlid	Africa
5	tetra	South America
6	loach	Asia