

# Hash indexing, index selection

Rasmus Pagh

# Today's lecture

- Morning session: Hashing
  - Static hashing, hash functions
  - Linear hashing
  - Newer techniques:  
Buffering, two-choice hashing
- Afternoon session: Index selection
  - Factors relevant for choice of indexes
  - Rules of thumb; examples and counterexamples
  - Designing the "best index" (at least in some cases)
  - Exercises



# What data in index?

- At least three possibilities:
  - 1) Record of key (only primary index)
  - 2) Key and pointer to record of key.
  - 3) Key and list of pointers to the records containing the key (for non-unique keys).
- For simplicity, we consider the case where there is the same number of keys ( $B$ ) in every disk block.
  - Case 1 with fixed length records.
  - Case 2 with fixed length keys.



# Static external hashing

- Hash table:
  - Array of  $\mathbf{R}$  disk blocks ( $\neq$  notation in RG.)
  - Can access block  $\mathbf{i}$  in 1 I/O, for any  $\mathbf{i}$ .
- Hash function  $\mathbf{h}$ :
  - Maps keys to  $\{0, \dots, \mathbf{R}-1\}$ .
  - Should be efficient to evaluate (0 I/Os).
  - Idea:  $\mathbf{x}$  is stored in block  $\mathbf{h}(\mathbf{x})$ .
- Problem:
  - *Dealing with overflows.*
  - Standard solution: Overflow chains.

# Randomized hash functions

Another approach (not mentioned in RG):

- Choose **h** at random from some set of functions.
- This can make the hashing scheme behave well regardless of the key set.
- E.g., "universal hashing" makes chained hashing perform well (in theory and practice).
- Details out of scope for this course...

# Analysis, static hashing

- Notation:
  - R blocks in hash table
  - Each block in the hash table can hold B keys.
- Suppose that we insert  $N = \alpha RB$  keys in the hash table ("fraction  $\alpha$  full", "load factor  $\alpha$ ").
- Assume  $\mathbf{h}$  is truly random.
- Expected number of overflow blocks:  
 $(1 - \alpha)^{-2} \cdot 2^{-\Omega(B)} R$  (proof omitted!)
- Good to have many keys in each bucket (an advantage of secondary indexes that store only pointers to records).
- Should keep  $\alpha$  away from 1. (How?)



## Sometimes, life is easy

- If  $B$  is sufficiently large compared to  $N$ , all overflow blocks can be kept in internal memory.
- Lookup in 1 I/O.
- Update in 2 I/Os.



# Too many overflow chains?

Can have too many overflow chains if:

- The hash function does not distribute the set of keys well ("skew").
  - Solution 1: Choose a new hash function.
  - Solution 2?: Overflow in main memory.
- The number of keys in the dictionary exceeds the capacity of the hash table.
  - Solution: Rehash to a larger hash table.
  - Better solution: ?
- There are many duplicate values.
  - No fix needed.





## Doubling the hash table

- For simplicity, assume  $R$  is a power of 2. Suppose  $h$  is a hash function that has values of "many" (e.g. 64) bits.
- We can map a key  $x$  to  $\{0, \dots, R-1\}$  by taking the  $\log R$  least significant bits of  $h(x)$ .
- Suppose that the hash table has become too small:
  - Want to double the size of the hash table.
  - Just consider one more bit of  $h(x)$ .

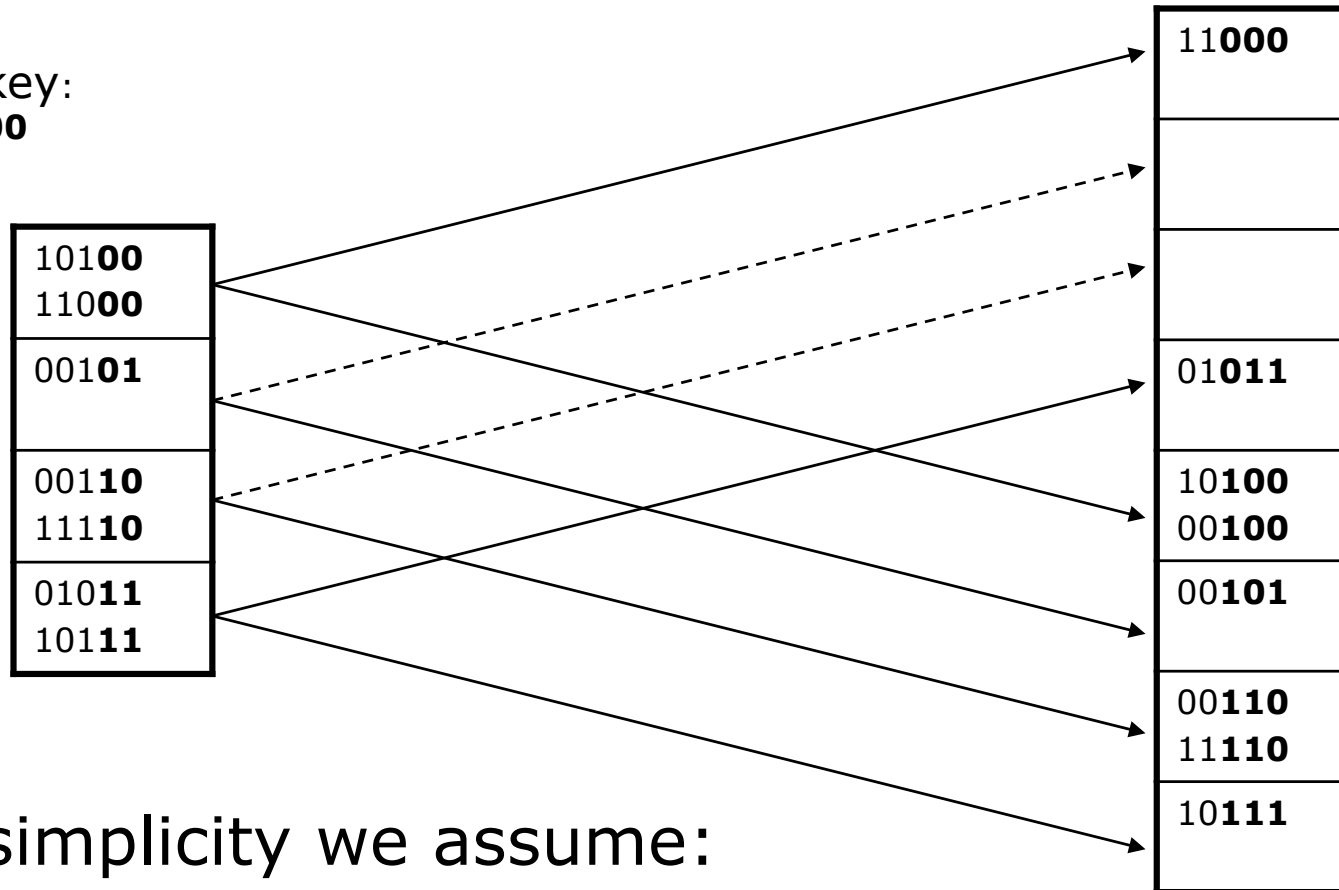
## Doubling the hash table, cont.

- Suppose  $h(\mathbf{x})=0111001$  (in binary) and the hash table has size 16.
- Then  $\mathbf{x}$  is stored in block number 1001 (binary).
- After doubling to size 32,  $\mathbf{x}$  should be stored in block 11001.
- Generally, all keys in block 1001 should be moved to block 01001 or 11001.
- **Conclusion:** Can rehash by scanning the table and split each block into two blocks.



# Doubling, example

New key:  
00100



For simplicity we assume:

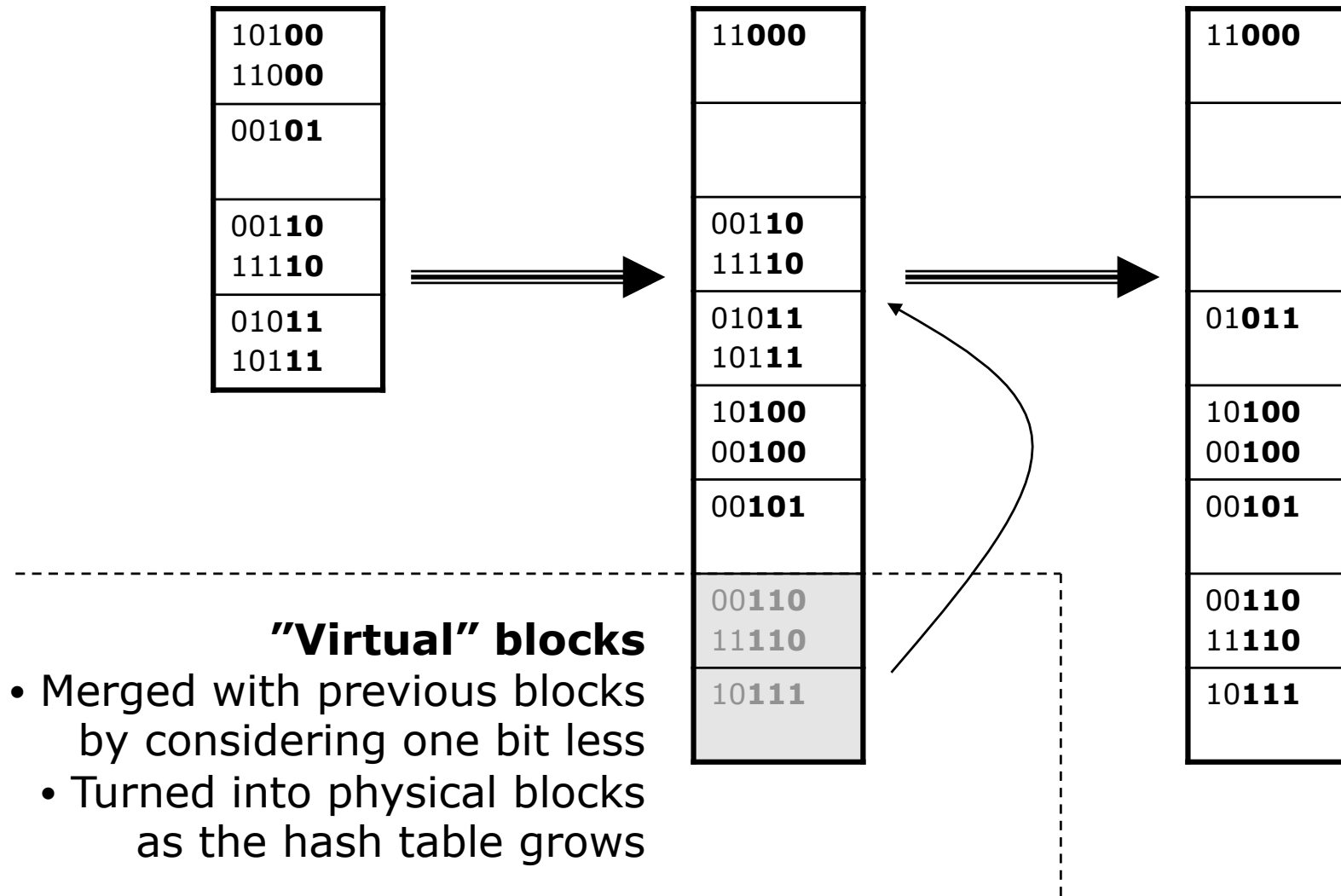
- No overflow chains
- $h(x)=x$

# Problem session

- Find some possible disadvantages of the doubling strategy. Consider:
  - Space usage vs overflows
  - System response time
  
- **Next:** An alternative that addresses some of the disadvantages of doubling.



# Linear hashing



# Linear hashing - performance

## The good:

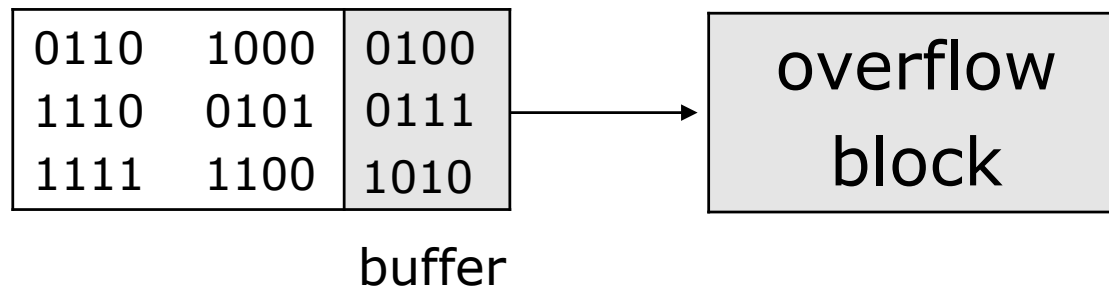
- Resizes hash table one block at a time:  
Split a block or merge two blocks.
- Cost of resize: 3 I/Os. Cheap!

## The bad:

- Increasing size of hash table may not eliminate any overflow chain.
- Uneven distribution of hash values; works best for relatively low load factors, 50-80%.  
(But variants of linear hashing improve this.)
- No worst-case guarantee on query time.

# Buffering

- Same trick as in buffered B-trees: Don't do updates right away, but put them in a buffer.



- Advantage: Several keys moved to the overflow block at once.
- Disadvantage: Buffer takes space.
- Details in [JensenPagh07].

# Two-choice hashing

- **Idea:**
  - Use two hash functions,  $h_1$  and  $h_2$ .
  - $x$  is stored in either block  $h_1(x)$  or  $h_2(x)$ , use two I/Os for lookup.
  - When inserting  $x$ , choose the least loaded block among  $h_1(x)$  and  $h_2(x)$ .
- Can be shown that overflow probabilities are much smaller than with one function, especially when  $B$  is small.
- If two disks are available, the 2 I/Os can be done in parallel.





## Tip: Hash indexing in Oracle

It is also possible to organize a table in a clustered hash index. First, create the hash table (called a “cluster”) using

```
CREATE CLUSTER <hashtablename>(<key>)  
HASHKEYS <hashtablesize>,
```

and then, when creating the table, specify that it is to be stored in the hash table:

```
CREATE TABLE <tablename>(<schema>)  
CLUSTER <hashtablename>(<key>).
```

The size of the hash table is fixed, i.e., the hash table will not grow or shrink

# Today's lecture, part 2

- Index selection
  - Factors relevant for choice of indexes
  - Rules of thumb; examples and counterexamples
  - Designing the "best index" (at least in some cases)
- Exercises

# Proactive vs reactive

- Proactive: Analyze, and try to anticipate need for indexes.
- Reactive: Identify performance problems in a running system, and try to resolve them (possibly using indexing).
- Today (and in most of this course) we focus on *proactive* index design.



# Workload

- The workload (mix of operations to be carried out by the DBMS) has a large influence on what indexes should be created in a database.
- Other factors are:
  - the data in relations, and
  - the query plans produced by the DBMS.



# Rules of thumb

- Rules of thumb can be used to guide thinking, and as a checklist.
- Are often valid in most cases, but there are always important exceptions.
- Quote from SB:

The point of the example is that the tuner must understand the reason for the rule

- You don't yet have the entire picture (query optimization, concurrency), but we can start reasoning about rules anyway.



# Rule of thumb 1:

## Index the most selective attribute

- Argument: Using an index on a selective attribute will help reducing the amount of data to consider.
- Example:  

```
SELECT count(*) FROM R  
WHERE a>'UXS' AND b BETWEEN 100 AND 200
```
- Counterexamples:
  - Full table scan may be faster than an index
  - It may not be possible/best to apply an index.



## Rule of thumb 2:

### *Cluster the most important index of a relation*

- Argument:
  - Range and multipoint queries are faster.
  - Usually sparse, uses less space.
- Counterexamples:
  - May be slower on queries “covered” by a dense index. (More on this later.)
  - If there are many updates, the cost of maintaining the clustering may be high.
  - Clustering does not help for point queries.
  - Can cluster according to *several* attributes by duplicating the relation!



# Tip: Clustered indexing in Oracle

A `CREATE INDEX` statement will create an unclustered B-tree index. The primary key of a relation is automatically indexed. If one wants to cluster the primary index, this must be specified when the relation is created, by using the keywords `ORGANIZATION INDEX` immediately after the relation schema.

- To cluster according to a *non-unique* attribute A, declare a composite primary key (A,P), where P is a unique key.



## Rule of thumb 3:

Prefer a hash index over a B-tree if point queries are more important than range queries

- Argument:
  - Hash index uses fewer I/Os per operation than a B-tree.
  - Joins, especially, can create many point queries.
- Counterexamples:
  - If a real-time guarantee is needed, hashing can be a bad choice.
  - Might be best to have **both** a B-tree and a hash index.



# Aside: Hashing and range queries

RG page 371:

Hash-based indexing techniques cannot support range searches, unfortunately.

- **But:** they can be *used* to answer range queries in  **$O(1+Z/B)$**  I/Os, where  $Z$  is the number of results. (Alstrup, Brodal, Rauhe, 2001; Mortensen, Pagh, Patrascu 2005)
- Theoretical result on external memory (why?)  
- and out of scope for DBT.

# Problem session

- Setting:
  - we have  $2^{20}$  tuples in a primary index
  - tuples take the space of 4 keys,
  - the space for a pointer is small compared to the space of a key
  - internal memory has space for  $M=2^{16}$  keys.
- Consider the search time of B-trees and extendible hashing in two cases:
  - Case A:  $B=4$  (i.e., 4 tuples/block).
  - Case B:  $B=2^6$ .

## **Rule of thumb 4:** **Balance the increased cost of updating with the decreased cost of searching**

- Argument: The savings provided by an index should be bigger than the cost.
- Counterexample:
  - If updates come when the system has excess capacity, we might be willing to work harder to have indexes at the peaks.
- If buffered B-trees are used, the cost per update of maintaining an index may be rather low. Especially if small degree trees are used.

## Rule of thumb 5:

**A non-clustering index helps when the number of rows to retrieve is smaller than the number of blocks in the relation.**

- Argument: In this case it surely reduces I/O cost.
- Counterexample:
  - Even for a non-clustered index, the rows to retrieve can sometimes be found in a small fraction of the blocks (e.g. salary, clustered on date of employment).



## **Rule of thumb 6: Avoid indexing of small tables.**

- Argument: Small tables can be kept in internal memory, or read entirely in 1 or 2 I/Os.
- Counterexample:
  - If the index is in main memory, it might still give a speedup.

## Rule of thumb 7:

***A covering index for a query will speed it up***

- Argument: The index will contain less data than the base table, allowing a faster scan of all data needed.
- Counterexamples:
  - If the table is vertically partitioned, a similar speedup can be achieved.
  - A vertically partitioned relation may have *several* indexes that can be used to answer the query (e.g. an index to select and an index to join).



## Next: Deriving the “best index”

- Proposal from Lahdenmäki and Leach, 2005.
- Focuses on select on a single table (possibly followed by a join).
- Assumes indexing is done with B-trees.
- Identifies two candidates, A and B
  - Further investigation may be needed to choose among them.



# The three stars

(B-tree) indexes may be evaluated on what "stars" they have, relative to a particular query.

- Star 1: The columns on which there is an equality comparison with a constant are the first columns of the index.
- Star 2: Results of every query is stored in the order given by ORDER BY.
- Star 3: Includes all columns mentioned in the SELECT.

# Candidate A

Choose columns of the index like this:

- First, the columns with an equality condition, e.g., `A=42`. (Star 1)
- Then, the most selective column with a range condition (`B BETWEEN 1 AND 10`).
- Then, remaining columns of the query. Let any remaining columns of an `ORDER BY` come first, in proper order.
  - Specify `DESC` for decreasing order attributes.

Has Star 1 and 3, but may not have 2.



## Candidate B

Choose columns of the index like this:

- First, the columns with an equality condition, e.g.,  $A=42$ .
- Then, remaining columns of the query. Let any remaining columns of an ORDER BY come first, in proper order.
  - Specify DESC for decreasing order attributes.

Has all stars, but does not make use of range conditions.



# Problem session

- Consider the following queries:
  1. 

```
SELECT id, fname FROM R
WHERE lname=:v1 AND city=:v2
ORDER BY fname
```
  2. 

```
SELECT id, fname FROM R
WHERE lname BETWEEN :v1 AND :v2
AND city=:v2
ORDER BY fname
```
- What are the Candidate A and B indexes? What stars do they qualify for?



# Sharing indexes

- In reality, several queries may share the same index
- It is not necessarily the best for any of them!
- Essentially, before adding an index one should ask if an existing index would suffice
  - Possibly, extra columns could be added to allow index-only query plans.



# Conclusion

- Indexing is a complicated business!
- Understanding the various index types and their performance characteristics, as well as the characteristics of the database at hand and its workload allows informed indexing decisions.
- Rules of thumb can be used to guide thinking.
- More complications to come!



# Exercises

Hand-outs:

- *Choosing an index.*
  - Questions a), b), and c).
- *Representation of relations*
  - Question d

(on handout from last week).