



This work is licensed under a Creative Commons Attribution 4.0 International License

CMPUT391

Database Access Methods

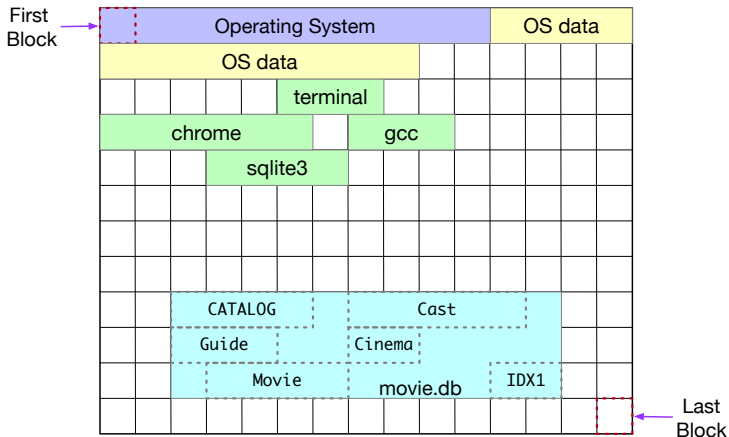
Instructor: Denilson Barbosa

University of Alberta

October 12, 2023

Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

How is the data laid out in the storage layer?



These notes are concerned with how the DBMS stores a database on disk and the cost of retrieving data.

Answering a query requires pulling data from disk to registers (via RAM) so that the CPU can compute the answer.

Some queries require the DBMS to read all tuples from a table:

```
SELECT AVG(imdb) FROM Movie
```

Most queries, however, require far fewer tuples.

For example, only tuples with 'Bill Murray' and 'Garneau' are needed here.

```
SELECT g.start  
FROM Guide g, Cast c  
WHERE g.film=c.title AND g.year=c.year  
      AND g.theater='Garneau'  
      AND c.actor='Bill Murray'
```

These slides look into methods for storing and/or indexing the data that the database administrator can use to speed up the (queries in the) application.

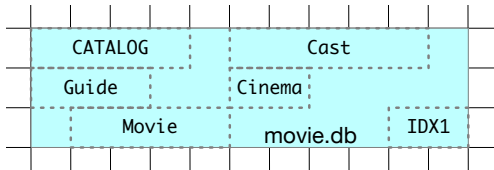
Introduction

Database “files”

Most DBMSs store an entire database in a single file (as seen by the Operating System)

- recall OS files are collections of blocks of persistent storage

Logically speaking, however, that file is broken down into many components, which in these notes we call “database files”.



Many DBMSs will grab many storage blocks from the Operating System (and leave them empty), to make sure there will be space for growth over time.

The database catalog

The catalog contains the *schemata* of all tables, the description of all indexes and triggers. On multiuser systems, it also contains all users and their privileges.¹

The catalog is used all the time: to parse SQL queries, to validate updates, etc.

Often the DBMS will bring the entire catalog to memory as part of the startup process.

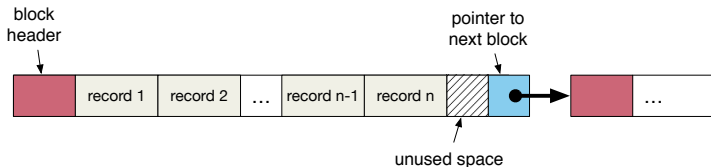
In most systems, the catalog is actually represented as a collection of tables, which the database administrator can inspect using SQL queries.

¹ Check out the `.schema` and `.dump` commands in SQLite.

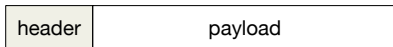
Heap files

The most common kind of DBMS file, called a heap file, is just a chain of blocks, with each block containing several **records**.

Each record represents a database object (e.g., a whole tuple).



Records have a header (metadata) and a payload (data):



In a *tuple-oriented store*, each record in a “table file” corresponds to a whole tuple².

In this case, the header typically contains:

- A pointer to the catalog—where the data types of the attributes in the tuple are described.
- A timestamp of the last update.
- A flag indicating if the record is valid (or has been deleted).³

²Except for **BLOB** (Binary Large Objects) values like images, sound, etc., in which the record has a pointer to the actual **BLOB** object.

³It is much cheaper to implement deletions by just marking tuples this way.

If the lengths of all attributes in a tuple are defined in the schema, the DBMS can use **fixed-length records**.

- In this scheme, every field of the record lies at a constant *offset* from the address where the record is loaded in memory.

Example:

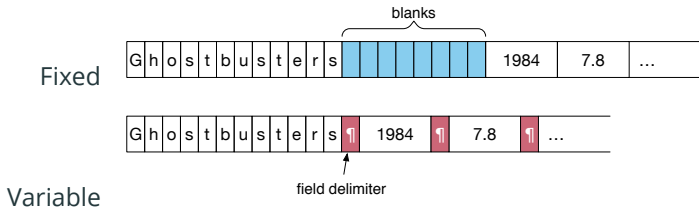
```
CREATE TABLE Movie (title CHAR(20), year INT, imdb FLOAT,  
    director CHAR(20), PRIMARY KEY (title, year),  
    CHECK(imdb >= 0 AND imdb <= 10));
```

The record would look like:

	title	year	imdb	director
header	20 x sizeof(char)	1 x sizeof(int)	1 x sizeof(float)	20 x sizeof(char)

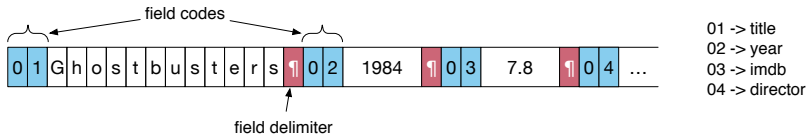
Fixed-length records can be wasteful for **NULL** values and textual attributes (e.g., names, addresses, etc.) because of variability in the length of actual values.⁴

With **variable-length** records, instead of “padding blanks” special codes as acting as *field delimiters* are used.



⁴SQL has data types for varying-length textual data like VARCHAR and TEXT.

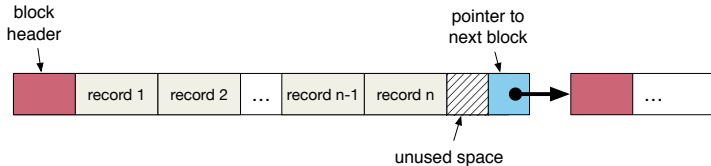
If NULL values are common in the data it is best to use both field delimiters *and* **field codes**; in this way the record contains only the fields whose values are not missing:



This kind of record can be used for non-relational data too, like semi-structured or XML.

Packing records into buffers/blocks

Regardless of the kind of record, most stores pack as many whole records into a block as possible. Space that is smaller than a record is *left unused*.



It is possible to allow a record to *span* consecutive blocks, but the trouble is often not worth it.

Fixed-length records:

- Fixed number of records per block: $\left\lfloor \frac{\text{block size}}{\text{record size}} \right\rfloor$.
- Space wasted on “blanks”.
- Records and fields start at known *offsets* within the block.

Variable-length records:

- Variable number of records per block.
- Space “wasted” on field separators.
- Need to scan the whole block to find records and fields.
- Can be used to store non-relational, semi-structured data.

ASIDE: word-aligned data structures

Once a record is loaded in memory, a processor in the CPU can get to a field by manipulating offsets or scanning through the record.

However, some CPUs allow reading/writing entire **words**⁵ (4 or 8 bytes, depending on the architecture) only.

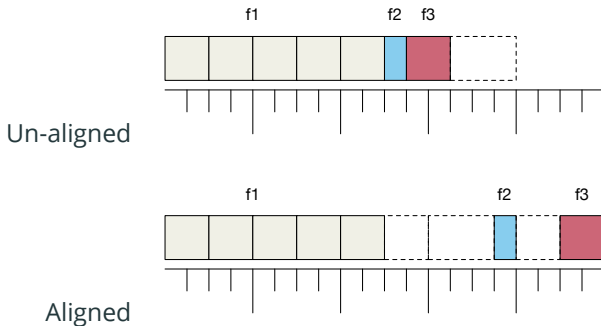
When this happens, either the storage stack aligns the fields, which is wasteful,⁶ or it contains code to properly “encode” and “decode” fields into words.

⁵https://en.wikipedia.org/wiki/Data_structure_alignment

⁶Some CPUs, on the other hand, allow access to individual bytes inside each word.

Example record:

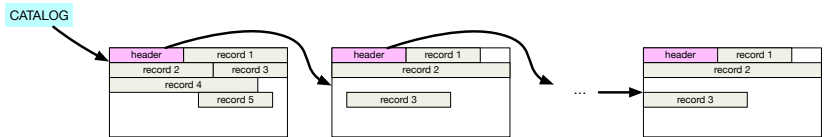
- f1: 5 UTF-16 (2 bytes) characters
- f2: 1 Boolean value (1 bit)
- f3: 1 smallint (2 bytes)



Heap and Sequential Files

Heap Files: Implementation and Access Cost

Heap files are similar to singly-linked lists, which are easy to maintain.



All the DBMS needs to record (in the Catalog) is a pointer to the first block in the file.

Each block has its own header for metadata, including, among other things, links to the next block in the chain.

Access costs

Recall from CMPUT204 that we measure computational costs by in terms of functions that bound **how many operations** the algorithm requires to complete.

For algorithms dealing with data structures in memory, we typically count accesses to memory.

Computational Cost units

Because I/O is so slow, we measure the *cost* as a function of **how many disk blocks need to be read or written** to perform the operation.

Which kinds of access are supported?

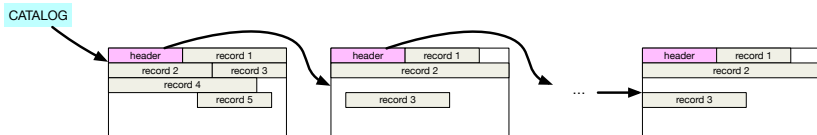
Inserting a new record.

Searching for a record, given a search key.

- Example: find movies with `director='Ivan Reitman'`

Deleting or **updating** a record.

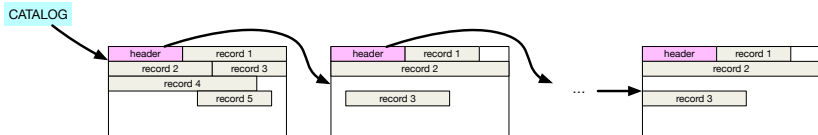
Note that before we can update or delete a record, we need to first *find* it. The costs for these operations in the following slides will include the search cost.



Inserting a new record in a Heap File:

- If there is room in the first block, add the record there.
- Otherwise, allocate another block to the file, add the record there, and link the new block to the previous first block.

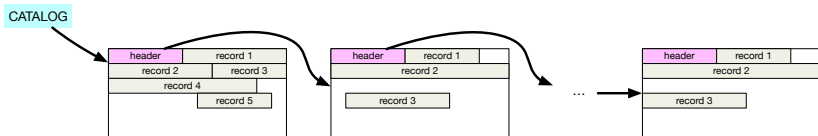
The **cost** of inserting a new record is $O(1)$



Searching for a record in a Heap File with N in blocks.

- Traverse all blocks, starting from the first block, following pointers until the record is found (or we exhaust the last block)

The **cost** of finding a record is $O(N)$.



Deleting or updating a record in a Heap File with N blocks:

- Find the record first (cost $O(N)$), and modify it, or remove it from the block.
- For deletions: if the block becomes empty, link the previous block to the next in the chain (unless the deletion happens in *last* block).

The **cost** of deleting a record is $O(N)$ ⁷.

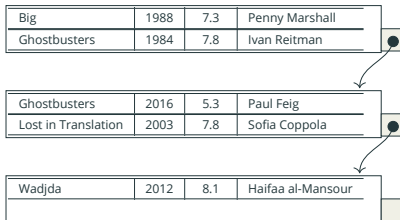
⁷Again, because of the search step.

Sequential Files

Sequential file are chains of blocks in which the records are *sorted* on a predefined set of attributes, both *within* and *across* blocks.

Tables with primary keys are always stored using sequential files with the primary key as the sorting attributes.

Example: sequential file for table `Movie` sorted by primary key (title, year).



Access cost of Sequential Files

Searches and deletions are the same as for heap files.

For **insertions** on a file with N blocks, we now need to:

- (1) Find the block where to insert the new record (at cost $O(N)$).
- (2) Insert the record if there is room, otherwise **rearrange the file**.

Two ways of rearranging the file:

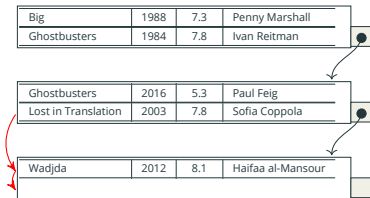
- moving existing records
- creating a new empty block

Insertion by rearranging records

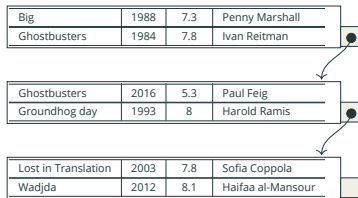
Example: inserting

Groundhog day	1993	8	Harold Ramis
---------------	------	---	--------------

Before



After



In the worst case scenario, need to keep moving records all the way to the end of the file.

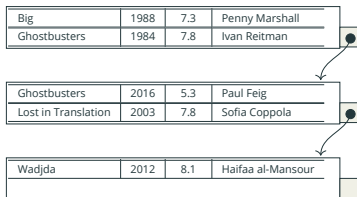
cost: $O(N)$ to rearrange the records in the file;
 $O(N)$ to find the block where to make the insertion.

Insertion by creating a new block

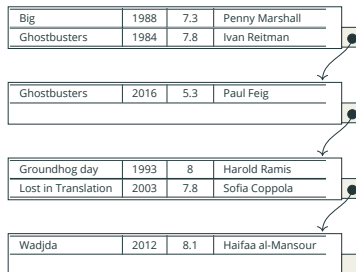
Example: inserting

Groundhog day	1993	8	Harold Ramis
---------------	------	---	--------------

Before



After



cost: $O(1)$ to modify the file;
 $O(N)$ to find the block where to make the insertion.

What about **updates** to a record in a sequential file?

Case #1: The update does not change the sort attributes in the record.

- In this case the record remains in the same block as it was⁸

Case #2: The update does change the sort attributes in the record.

- This case is best handled by a deletion of the “old” record (before the update) followed by the insertion of the “new” record (after the update).

⁸Unless, of course, the file uses variable length records and the update makes the record too large to fit in the block! In which case a new block must be added to the file.

Heap vs Sequential Files

Both are based on singly-linked lists.

Heap files are better for insertions, but all other operations cost the same in both. If N is the size (in blocks) of the file, then:

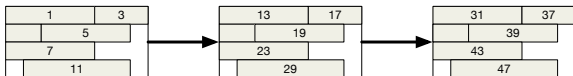
	Heap	Sequential
insertion	$O(1)$	$O(N)$
search	$O(N)$	$O(N)$
deletion	$O(N)$	$O(N)$
update	$O(N)$	$O(N)$

We need indexes!

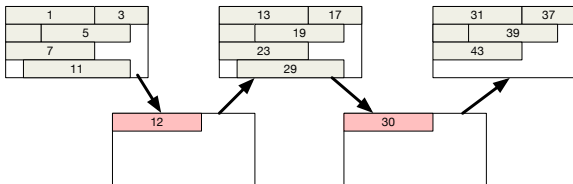
Indexes are needed to cut down the $O(N)$ time to find tuples inside the table file (heap or sequential).

Sequential files can have many quasi-empty blocks after many insertions... which wastes space and (more importantly) I/O operations.

Before:



After:



This can be fixed by **periodically** taking the database *offline* and re-building the file.

Indexing Concepts

Indexes Are Associative Data Structures

An index is an **associative data structure** (stored on disk) that allows the DBMS to find tuples that satisfy a predicate defined over a subset of the attributes in the tuple.

Index declarations are part of the SQL DDL:

```
CREATE INDEX name On Table(attr_1 [, attr_2 [, ... [, attr_n]]]);
```

The sequence of attributes **attr_1**, **attr_2**, ... , **attr_n** is called the index key (a.k.a., sort key, or search key).

The index **associates** every index key with a **pointer** to the tuple containing that key.

Indexes save time in answering queries by allowing the DBMS to skip tuples that do not satisfy predicates in the `WHERE` clause⁹.

```
CREATE INDEX IDX1 ON Movie(title);
```

Example:

```
SELECT director
FROM Movie
WHERE title='Ghostbusters' AND year=1984
```

Even if there are thousands of movies, the DBMS can use `idx1` to quickly find only those with “Ghostbusters” in the title (and then check the condition on year).

⁹DBAs always look at the query workload to look for opportunities to add more indexes.

Index maintenance:

```
CREATE INDEX IDX1 ON Movie(title);
```

The DBMS keeps the index **synchronized** with the table:

- Every tuple of `Movie` has a corresponding entry in `IDX1`.
- The DBMS updates the index **after every update** (insertion, deletion or update of tuples) to the table.

Flat Indexes

The next slides develop the basic ingredients of the B+ tree index, which is prevalent in modern DBMSs. We start with “**flat indexes**”, corresponding to the lower level of a B+tree:

A **flat index** is a **sequential file** in which every record has two fields:

- the index **key**: one or more attributes of tuple t_i
- a **pointer** to t_i

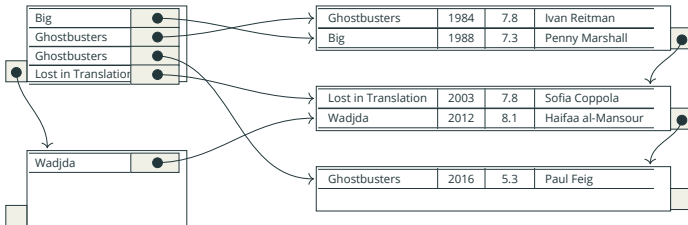
As we will see, **hierarchical indexes** such as the B+tree are built on top of flat indexes, and this is why we will spend some time understanding them.

Example flat index on attribute `title` of table `Movie`:

```
> CREATE INDEX IDX1 ON Movie(Title);
```

Index File

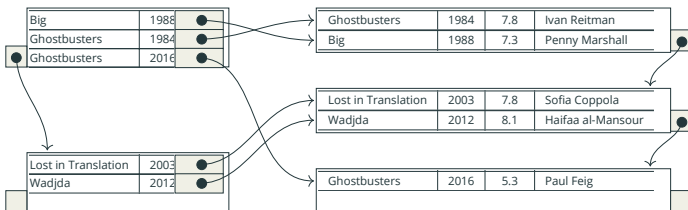
Data File



Index keys can be more than one column.

Example flat index on attributes `Title` and `Year`:

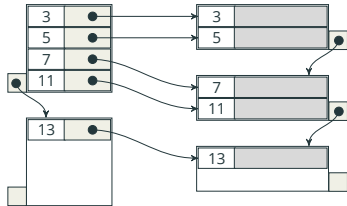
```
> CREATE INDEX IDX2 ON Movie(Title,Year);
```



The **order of the attributes in the key matters!** An index on `Title,Year` is very different than an index on `Year,Title`.

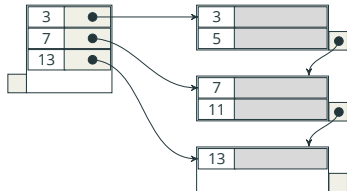
Terminology

A **dense index** has every index key in the table (meaning one pointer per tuple)



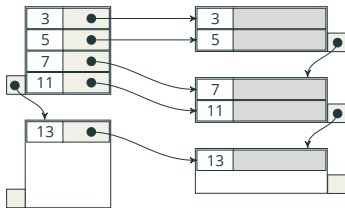
A **sparse index** has only some of the keys (and fewer pointers than tuples)

Normally, each key in the index points to a different block of the data



Primary Index

A **primary index** is a dense index on a sequential file *sorted by the same keys* as the index.



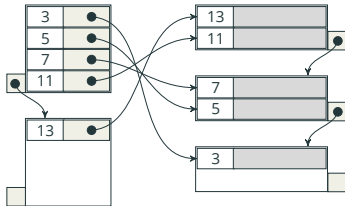
Primary indexes speed up enforcing primary key constraints.

Secondary Index

In a **secondary index**, the keys in the index file are not sorted in the same way as the records on the data file.

Secondary indexes are always dense.¹⁰

Indexes created on attributes that are not the primary key are secondary indexes.



¹⁰It should be obvious that a sparse secondary index would be useless.

How does an index save time?

Blocks of the index file are the same size as the blocks of the (sequential or heap) data file.

Because the **index records are smaller**, each block of the index covers more keys than a block of the data file.

Typically, a disk block can hold:

- a few hundred key/pointer pairs or
- dozens of tuples

Example: table with 1 million tuples; each disk block can hold:

- 50 tuple (records) or
- 200 key-pointer pairs

Sizes:

- data file: 20,000 blocks
- index file: 5,000 blocks

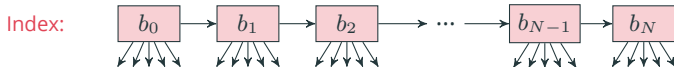
Binary search on an index?

Binary search for key k on a sorted list of keys **in memory**:

- find m , the key "in the middle" of the list
- stop the search if $m = k$;
- search recursively on the "lower half" if $k < m$
- search recursively on the "upper half", otherwise

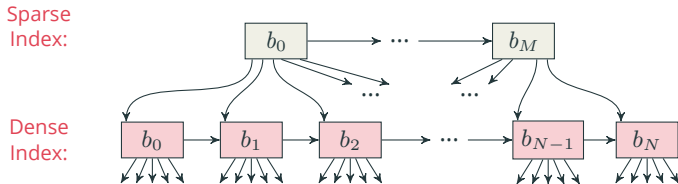
Binary Search does not work on an index file on disk

Binary Search requires $O(1)$ access time to *any* of the keys, which is not possible if the blocks of the index are on disk.



Multi-level indexes

One way to reduce the number of I/Os to search for a key in an index is to **stack** a sparse index over a dense index, of course, using the same index key.

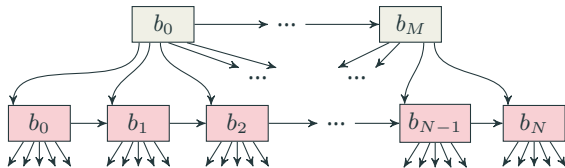


Each record in the sparse index contains the **smallest** key in each block of the index below.

- Therefore, every record in the sparse index corresponds to 200 records in the dense index “below”.

Sparse
Index:

Dense
Index:



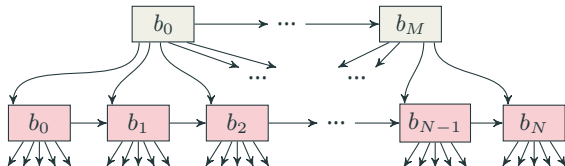
To find a record with index key $k = x$:

- (1) Search the sparse index for $\max(x') \leq x$.
- (2) Search the block of the dense index starting at key x' .
- (3) If the key is found, traverse the pointer to read the tuple from the table file.

How large will the sparse index be?

Sparse
Index:

Dense
Index:



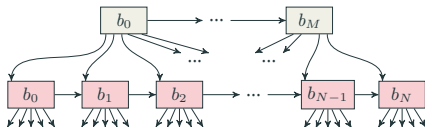
Example: table with 1 million tuples; each disk block can hold:

- 50 tuple (records) or
- 200 key-pointer pairs

We would have:

- data file: 20,000 blocks
- dense index: 5,000 blocks
- sparse index: **25 blocks**

Example: cost of finding a tuple in the table by searching for a key.



- data file: 20,000 blocks
- dense index: 5,000 blocks
- sparse index: 25 blocks

Cost of **retrieving** a tuple with key k :

- without any index: 20,000 I/Os
- with dense index only: 5,000 +1 I/Os
- with both indexes: 25 +2 I/Os

Access Costs With (Stacked) Flat Indexes

Recall from slide 26 that insertions, deletions and updates depended on the cost of *finding* the tuples in the table files first.

Assume table R has $T(R)$ tuples and is stored on M blocks on disk, and that we are searching for tuples satisfying a condition c .

Generally, the cost has two parts:

- (1) Finding the tuples in the index.
- (2) Retrieving each such tuple from the table (via a pointer traversal)

If the table is stored on a heap file, the number of pointer traversals is $\lceil s(R, c) \cdot T(R) \rceil$. Otherwise, it will be $\lceil s(R, c) \cdot M \rceil$.

The index cost is $\lceil (1 - s(R, c) \cdot K) + h - 1 \rceil$, where K is the size (in blocks) of the top-most flat index and h is the number of stacked indexes.

I/O cost to retrieve all tuples in R satisfying condition $c : k = x$:

(1) R on **heap file**, no index: M block reads

(2) R on **sequential file**, no index:

$$\lceil M \cdot (s(R, k < x) + s(R, k = x)) \rceil$$

If we have a dense flat index on k occupying N blocks:

(1) R on **heap file**: $\lceil N \cdot s(R, k \leq x) + s(R, k = x) \cdot T(R) \rceil$

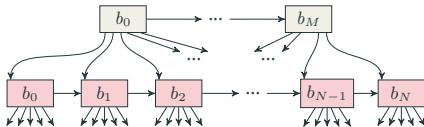
(2) R on **sequential file**: $\lceil N \cdot s(R, k \leq x) + M \cdot s(R, k = x) \rceil$

If we have stacked sparse/dense indexes on k occupying K and N blocks, respectively:

(1) R on **heap file**: $\lceil K \cdot s(R, k \leq x) + 1 + s(R, k = v) \cdot T(R) \rceil$

(2) R on **sequential file**: $\lceil K \cdot s(R, k \leq x) + 1 + M \cdot s(R, k = x) \rceil$

Every sparse index we add **further reduces the fraction** of the blocks from an index below that we need to scan:



Thus, the more levels of sparse indexes the better!

B+trees take this idea to the extreme, defining a self-balanced tree of index blocks in which:

- (1) the topmost level (the root) is a **single-block** sparse index
- (2) the leaves form a dense flat index
- (3) the levels in between are sparse indexes

Index maintenance

Any index is only useful if kept up-to-date after updates to the associated table.

Flat indexes are sequential files, and thus have the same update costs as in slide 26.

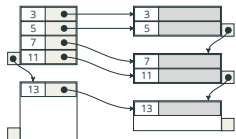
Updating sparse indexes (like the higher levels of a multi-level index) is no different: all we need to look for is if there is room in the right block.

The next few slides show examples of issues that arise when updating a flat index and its table.

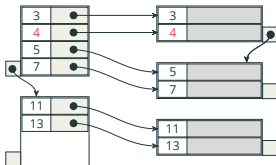
Example: inserting a tuple with index key

4. Assume the table is kept in a

sequential file for now.



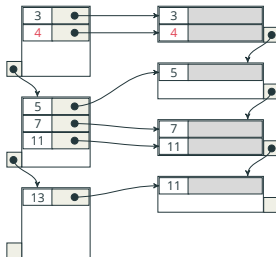
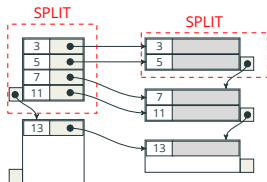
Option 1: Re-arrange records to use existing free space:



I/O intensive:

- may require moving $O(|R|)$ of tuples to either end of the file
- same for the index

Option 2: Split existing blocks or add new ones as needed.



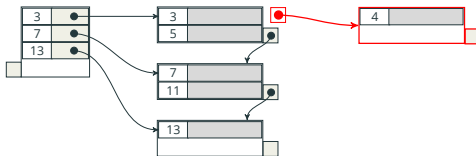
Pro: fixed number of blocks added/split; in other words, this strategy has $O(1)$ (constant) I/O cost

Con: too much empty space reduces the effectiveness of the index

With a sparse index, the DBMS can avoid modifying the index by *expanding* a data block with an **overflow** block:

Example: insertion of tuple with index key 4.

Logically, there is now *one* block with keys 3, 4 and 5.



Many overflow blocks can be used, forming an **overflow chain**.

With overflow blocks or chains the search time becomes non-uniform, as some blocks are larger than others.


During re-indexing, the DBMS removes all overflow chains and re-builds the index.

Deletions are, logically, the opposite of insertions. Which means that if we insert a tuple that causes the table file and the index file to grow, deleting that same tuple should cause both files to shrink.

But is that what the DBMS should do?

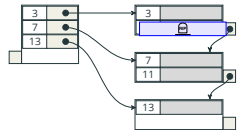
Performing deletions in this way incurs in more I/O operations.

And we may have to grow the files again in the future due to another insertion.

In practice, the best way is to mark the deleted records with “tombstone” marks ()^{RIP}, leaving the space available for future insertions.

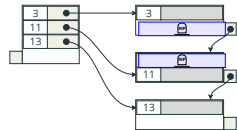
Example: deletion of tuple with index key 5.

A record with key between 3 and 7 can go where the record with 5 was.



Example: further deletion of tuple with index key 7.

Any record with index key between 3 and 11 can go on either block.



Updating a search key?

Update statements that modify the index key are handled like a deletion (of the original tuple) followed by an insertion (of a “new” tuple with the new index key).

After many insertions and deletions causing node splits or the use of tombstones, the data and index files can become **fragmented**, leading to sub-optimal I/O performance.

Re-indexing

Database administrators can set the DBMS to periodically re-build all data and index files, to redistribute the records across all blocks, removing largely unused blocks.

Range Searches

Index searches on ranges of values

Whether or not an index on attribute a helps answer $\sigma_{a>x}(R)$ depends on the kind of index and on how many tuples fall in the range $a > x$.

Selectivity of a predicate

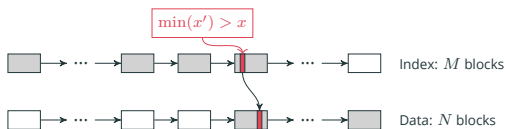
The **selectivity** of a predicate c for R , is the fraction of **tuples** in R that satisfy c .

$$s(R, c) = \frac{|\sigma_c(R)|}{|R|}$$

The slides on query processing discuss how to estimate selectivity.

Scenario 1: a primary flat index on a sequential file.

Assume the index has M blocks and the table file has N blocks.



Cost: $|I_{a < x}| + |R_{a > x}|$ block reads

$|I_{a \leq x}| = \lceil M \cdot (1 - s(R, a > x)) \rceil$: number of index blocks up until the first record with key x

$|R_{a > x}| = \lceil N \cdot s(R, a > x) \rceil$: number of data blocks starting at x

Example: 1 million tuples:

- data file: 20,000 blocks
- flat index: 5,000 blocks
- $s(R, a > x) = 0.2$

I/O Cost= index scan + table scan

$$\lceil 5,000 \cdot 0.8 \rceil + \lceil 20,000 \cdot 0.2 \rceil$$

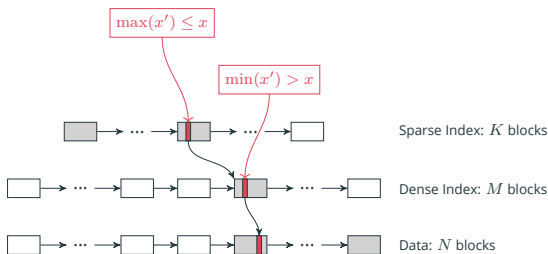
In practice, the DBMS decides whether or not to use the index based on an *estimate* of the selectivity of the predicate.

The actual number of tuples satisfying a selection condition depends on data distributions which can be quite complex. But the DBMS cannot “remember” everything about the data in memory. Instead, it keeps some descriptive statistics to perform its estimates.

Scenario 2: multi-level index with a sparse on top of the same dense index as before:

Example: 1 million tuples:

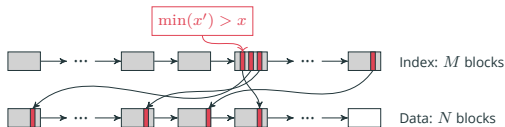
- data file: 20,000 blocks
- dense index: 5,000 blocks
- sparse index: 25 blocks
- $s(R, a > x) = 0.2$



I/O Cost= sparse index scan + **dense index scan** + table scan

$$\lceil K \cdot 0.8 \rceil + \mathbf{1} + \lceil N \cdot 0.2 \rceil$$

Scenario 3: (single-level) dense **secondary** index.



In this case, the DBMS has to read the entire index and perform a traversal, potentially to a different block, for each pointer out of the index satisfying the predicate.

Example: 1 million tuples:

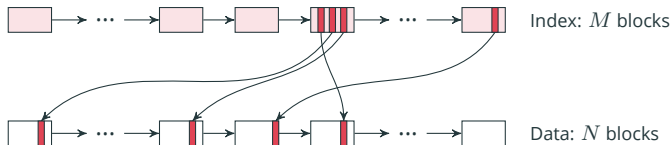
- data file: 20,000 blocks
- dense index: 5,000 blocks
- $s(R, a > x) = 0.2$

I/O Cost= index scan + pointer lookups

$$5,000 + \lceil 1,000,000 \cdot 0.2 \rceil$$

Bucket Files and Efficient Secondary Indexes

Secondary indexes are defined over attributes that are not part of the primary key. Therefore, the tuples in the file are **not sorted** according to the index keys:



Moreover, it is common for **many tuples share the same values** for these non-key attributes. For example, many movies have the same director.

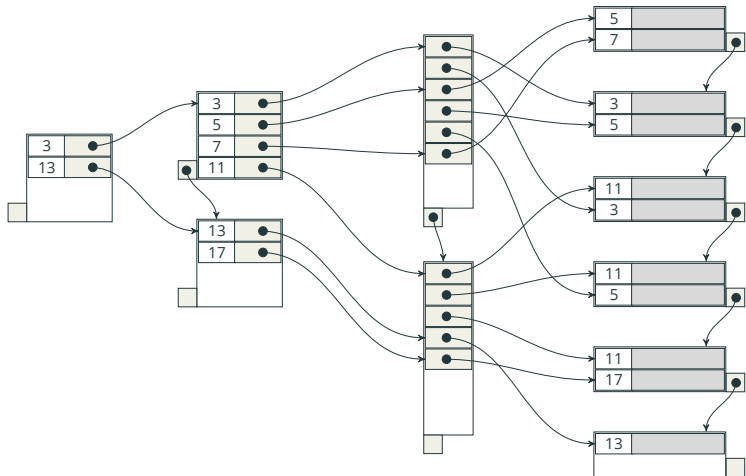
In such cases, the same key is repeated multiple times in the index, which can be avoided with the help of bucket files.

Bucket files for secondary indexes

Index File

Bucket File

Data File



Bucket files contain **tuple identifiers** (i.e., pointers) clustered according to the index key.

Bucket files save space when there are many duplicate index keys (which is common with secondary indexes), and allow the DBMS to quickly find (ids of) tuples based on an attribute value:

Algorithm 1 retrieving ids of tuples with $a = x$ with secondary index on a

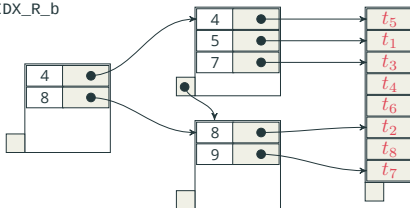
```
1: answer  $\leftarrow \langle \rangle$ 
2:  $p_1 \leftarrow$  pointer (to bucket file) associated with key  $x$  in the index
3:  $p_2 \leftarrow$  pointer associated with the next key in the index (or  $\langle \text{EOF} \rangle$  if  $x$  is last key)
4: open bucket file at position  $p_1$ ;
5: repeat
6:   append tuple id from current position to answer;
7:   advance pointer on bucket file
8: until reach  $p_2$ 
9: return answer
```

Example: table $R(a, b, c, d)$;

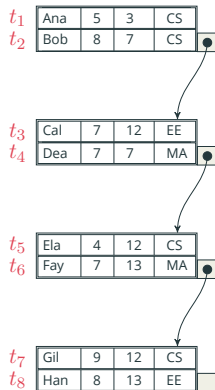
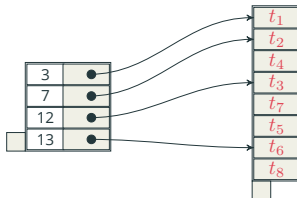
primary index on $R.a$;

secondary indexes on $R.b$ and $R.c$.

IDX_R_b



IDX_R_c



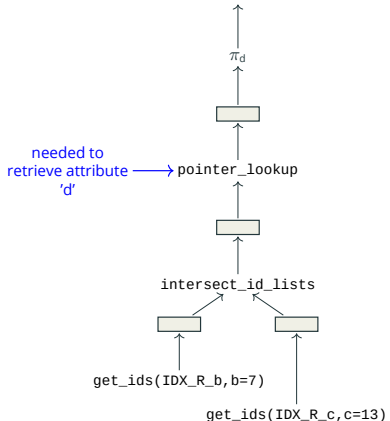
Complex selections with tuple ids

The DBMS can solve an arbitrary selection using lists of tuple ids that satisfy each predicate separately.

SELECT d
FROM R
WHERE b = 7 **AND** c = 13;

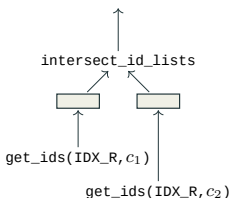
$\text{get_ids}(\text{IDX_R_b}, b=7) = \langle t_3, t_4, t_6 \rangle$

$\text{get_ids}(\text{IDX_R_c}, c=13) = \langle t_6, t_8 \rangle$



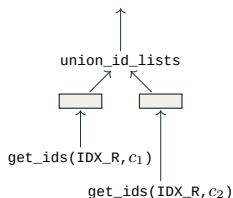
Conjunction $\sigma_{c_1 \wedge c_2}(R)$

list intersection



Disjunction $\sigma_{c_1 \vee c_2}(R)$

list union



Because tuple ids are **clustered** in the bucket files, this approach can lead to very fast answers!

Cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Example: $R(a, b, c, d, \dots)$; 1 million tuples:

- 1000 distinct values of attribute a ;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuple (records) or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- index on $R.a$: 5 blocks
- bucket file $R.a$: 2,500 blocks
- index on $R.b$: 1 block
- bucket file $R.b$: 2,500 blocks

Table scan: requires reading 20,000 blocks

Merging id lists:

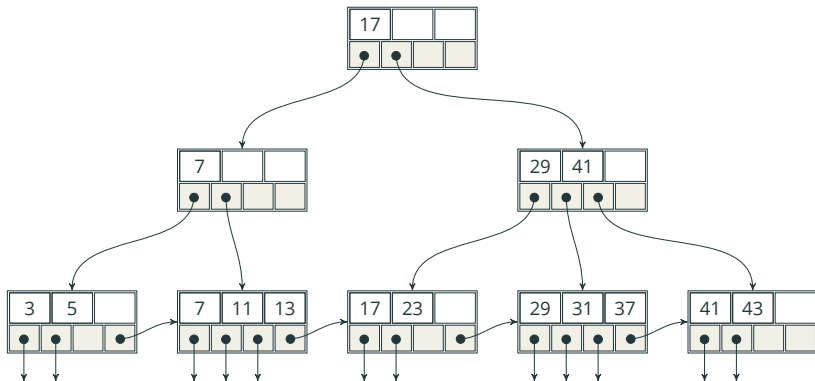
$$(5 + 1 + \lceil 2,500 \cdot s(R, a = k_1) \rceil + \lceil 2,500 \cdot s(R, b = k_2) \rceil + \min(s(R, a = k_1), s(R, b = k_2)) \cdot 1,000,000) \text{ blocks}$$

NOTE: Better asymptotic costs are possible, e.g., with B+trees.

B+ Trees

B+ tree

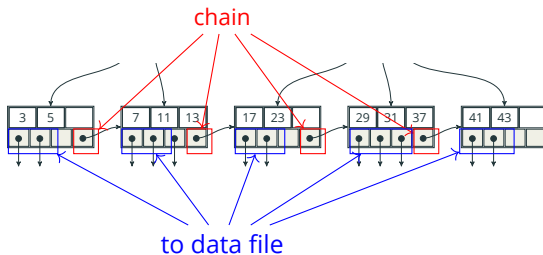
The B+ tree is a self-balancing search tree designed for secondary memory: **each node of the tree is stored on a separate disk block.**



Every node holds n keys and $n + 1$ pointers

In a leaf node:

- the first n pointers point to actual tuples in the data file
- the last pointer is used to form a **chain** of leaf nodes (useful for range queries)

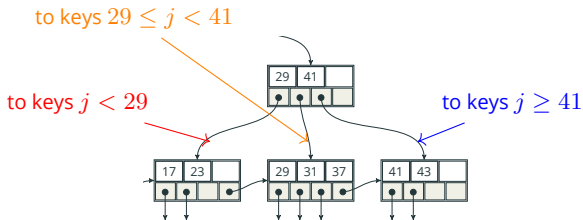


Every node holds n keys and $n + 1$ pointers

In an inner node with i keys the first $i + 1$ pointers are used.

A search for key j follows pointer:

$$\begin{cases} 0 & \text{if } j < k_0 \\ i & \text{if } k_{i-1} \leq j < k_i \\ i + 1 & \text{if } j \geq k_i \end{cases}$$



Every node holds n keys and $n + 1$ pointers

To avoid nodes becoming “too empty”, the minimum **occupancy** of a node should be:

- **inner node:** $\left\lceil \frac{n+1}{2} \right\rceil$ pointers

- **leaf node:** $\left\lfloor \frac{n+1}{2} \right\rfloor + 1$ pointers (to data file)

If the node occupancy goes below that (due to deletions), the node should be merged with adjacent ones¹¹

¹¹In practice, however, most DBMSs skip this and reclaim wasted space during re-indexing maintenance processes.

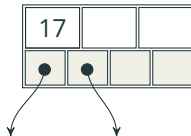
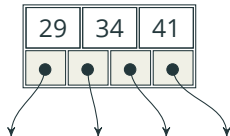
Example: occupancy with

$$n = 3$$

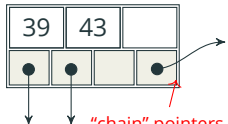
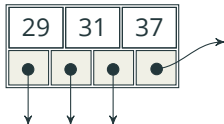
Full node

Minimum occupancy

Inner :

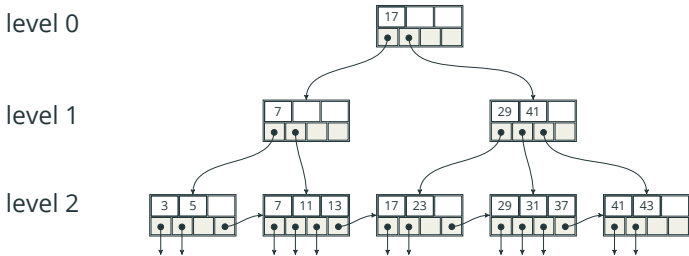


Leaf :



"chain" pointers
count EVEN IF
THEY ARE SET TO
NULL

B+ trees are balanced



The **level of a node** is its distance (in edges) from the root.

The **height of the tree** is the maximum level of any node.

In a B+ tree **all leaves are at the same level.**

Root nodes are special

Often, real B+ tree nodes (i.e., disk blocks) hold a few hundred pointers. Thus, until there are more than that many keys in the index, the “entire tree” is just one (root *and* also leaf) node.

	minimum		maximum	
	pointers	keys	pointers	keys
Inner (non-root)	$\left\lceil \frac{(n+1)}{2} \right\rceil$	$\left\lceil \frac{(n+1)}{2} \right\rceil - 1$	$n + 1$	n
Leaf (non-root)	$\left\lceil \frac{(n+1)}{2} \right\rceil + 1$	$\left\lceil \frac{(n+1)}{2} \right\rceil$	$n + 1$	n
Root	2	1	$n + 1$	n

NOTE when the tree is just the root node the minimum number of pointers is 1.

Inserting a new key into a B+ tree

B+ tree insertions are very similar insertions into binary search trees:

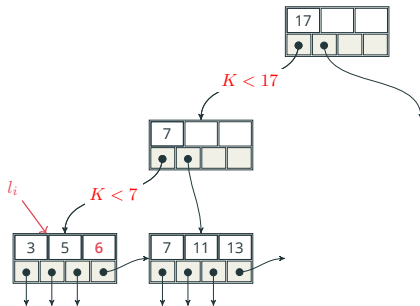
(1) find where the new key should go; (2) add the new key; (3) modify other nodes as needed.

- (1) **Search** for key K in the tree; that is, traverse the tree from the root until we locate the leaf node l_i where K should go.
- (2) If there is space in l_i , add key K ; **STOP**. Otherwise:
 - (a) create a new leaf l_{new} with K ;
 - (b) divide the keys in l_i among l_i and l_{new} so that both satisfy the minimum occupancy requirement;
- (3) link l_{new} to l_i and to its predecessor or successor; link l_{new} to the parent of l_i **recursively** inserting a new key in the parent.
- (4) If needed, recursively add and link parents nodes.

Example: insertion when there is room in l_i

Insert key $K = 6$

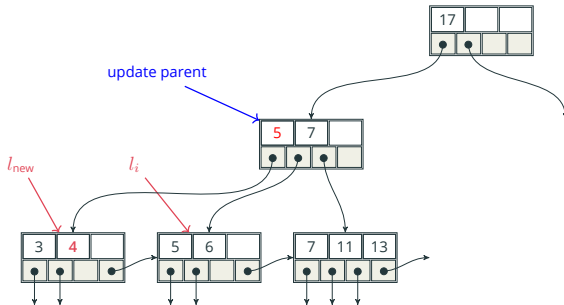
$n = 3$



Example: insertion causing a node split “to the left” of l_i

Insert key $K = 4$

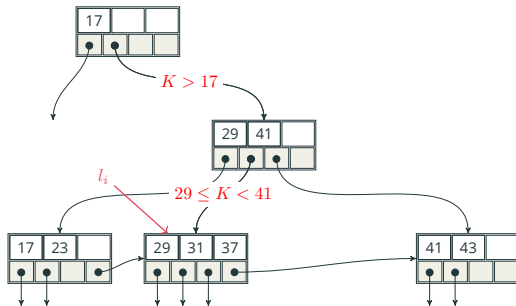
$n = 3$



Example: insertion causing a node split “to the right” of l_i

Insert key $K = 32$;

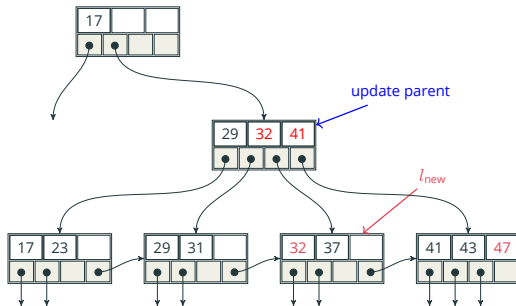
$n = 3$



Example: insertion causing a node split “to the right” of l_i

Insert key $K = 32$; and then insert key $K = 47$;

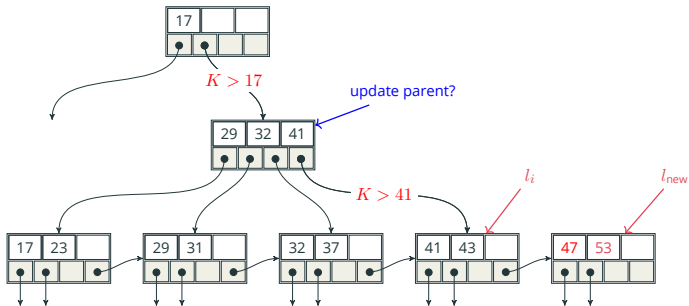
$n = 3$



Insertion leading to adding a new parent:

Insert key $K = 53$

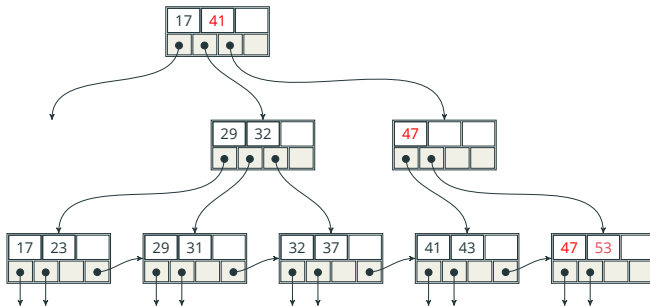
$n = 3$



Insertion leading to **adding a new parent**:

Insert key $K = 53$

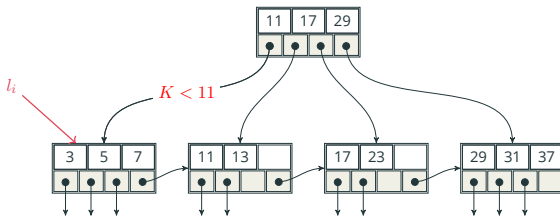
$n = 3$



Insertion leading to **adding a new root**:

Insert key $K = 2$

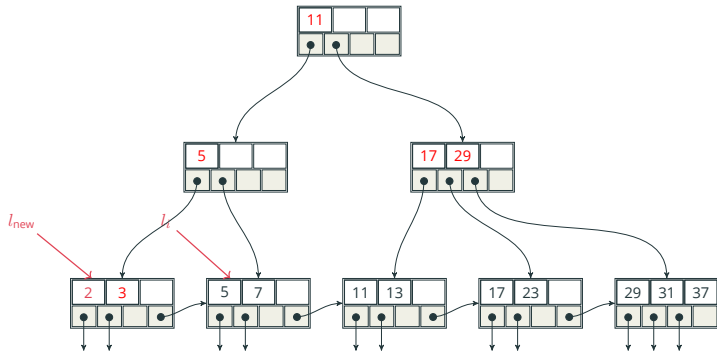
$n = 3$



Insertion leading to **adding a new root**:

Insert key $K = 2$

$n = 3$



B+tree Insertions: Wrap Up

Insertions have low cost.

In most systems, nodes can hold a few hundred keys, and after the tree grows to a reasonable size (usually 2 or 3 levels), only a small fraction of insertions require adding/modifying inner nodes.

The tree always grows “upwards”, by adding *parents*.

The worst case scenario of inserting into a tree T is bound by $O(\text{height}(T))$, which is practically a constant (compared to the number of keys—i.e., tuples in the database).

Deletions from B+ trees

When deleting a key from a B+ tree the main concern is ensuring the node occupancy **does not** go below the minimum allowed.¹²

If a deletion does not make the node occupancy too low, there is nothing further to do.

Dealing with a node with low occupancy (leaf or inner):

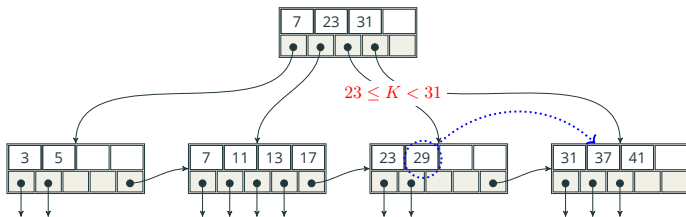
- **coalesce** with a neighboring node
- **move a key** from a neighboring node

¹²Recall that the more keys in a node (disk block) the *higher* the I/O efficiency.

Coalescing with neighbor node.

Delete key $K = 23$

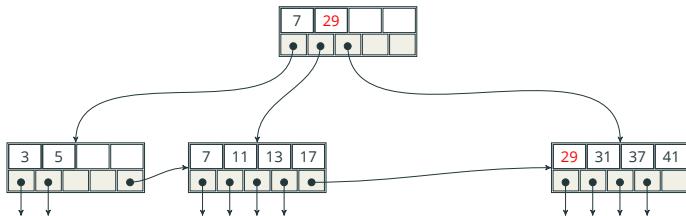
$n = 4$; min occupancy is 2 keys



Coalescing with neighbor node.

Delete key $K = 23$

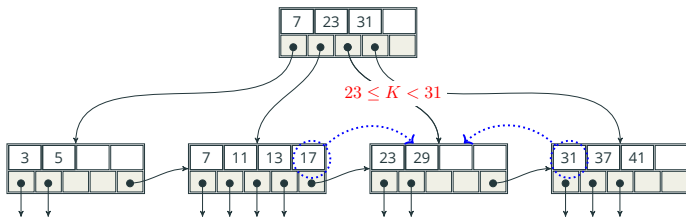
$n = 4$; min occupancy is 2 keys



Moving a key from a neighbor node.

Delete key $K = 23$

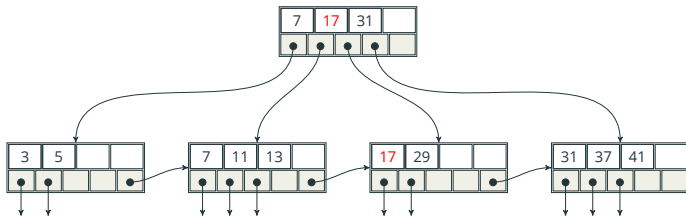
$n = 4$; min occupancy is 2 keys



Moving a key from a neighbor node.

Delete key $K = 23$

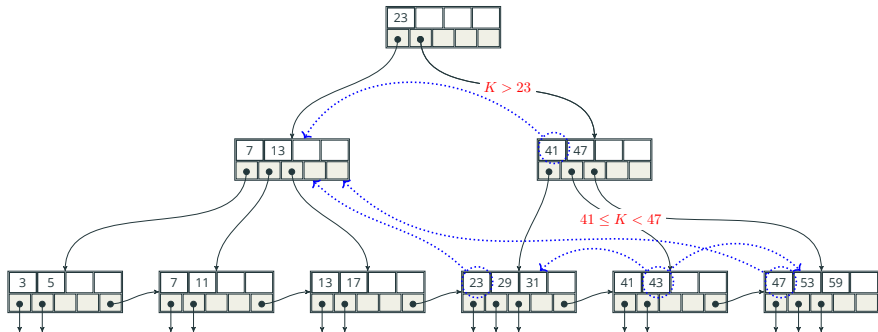
$n = 4$; min occupancy is 2 keys



Coalescing inner nodes.

Delete key $K = 41$

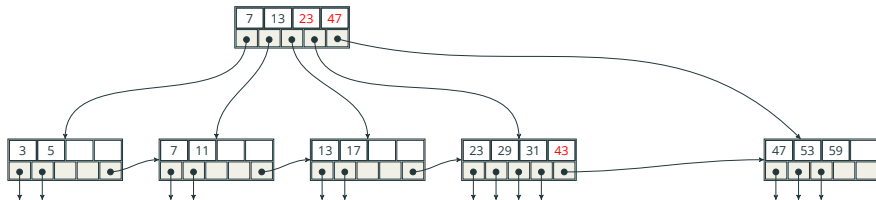
$n = 4$; min occupancy is 2 keys




Coalescing inner nodes.

Delete key $K = 41$

$n = 4$; min occupancy is 2 keys



B+tree Maintenance in Practice

Like with flat indexes, in practice, deletions in B+ trees are handled by marking the records at the leaf nodes with “tombstone” markers (). When the database goes offline for maintenance, the index is rebuilt and the tombstones are removed.

The overhead of removing nodes and rearranging pointers to disk blocks is not worth the potential savings they would introduce.

B+tree: cost summary

B+trees can be seen as

- a dense flat index at the leaf-level which supports range searches
- a hierarchical sparse index over that flat index, with height H

Searching for a key requires reading $O(H)$ disk blocks. In practice, B+trees with 3 levels can index a very large range of search keys.

Insertions may require adding new leaf nodes and/or new inner nodes, all the way up to the root, at cost $O(H)$ where H is the height of the tree.

Deletions cost the same as insertions, but many DBMSs just use “tombstone” markers instead for simplicity.

How useful are B+ trees in practice?

Indexes are used to implement primary and foreign keys and to speed up the evaluation of predicates in a **WHERE** clause of a query. Without indexes, all of these operations would require reading $O(N)$ blocks of a “table file” with N blocks.

With B+trees, except for range queries, all costs are reduced to $O(H)$, where h is the height of the B+tree, which in turn is $O(\log_k N)$, where N is the number of blocks in the table file and k is the number of pointers in each B+tree node.

In complexity terms, **indexes trade time for space**: they require several disk blocks (space), but save (a lot) of time in accessing tuples. Because secondary storage is ever more inexpensive, indexes are a great idea.

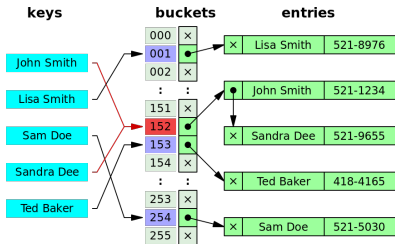
Hash Files

Hash-based Associative Data Structures

A hash table¹³ is a data structure that maps **keys** to **values**.

A DBMS can use a hash table (a.k.a a hash file) to store tuples of a table on which a primary key has been defined.

In this setting, the **keys** are the attributes of the primary key, and the **values** are the entire tuples themselves.



¹³See https://en.wikipedia.org/wiki/Hash_table.

Hash tables are suitable for implementing primary keys and foreign keys efficiently.

- They provide *Expected* $O(1)$ cost for finding if a key is already in the table, which allows the DBMS to check violations to key constraints.

Hash tables are also suitable for implementing joins on primary/foreign key pairs.

Hashing is also used in a DBMS to implement a set data structure, needed for duplicate elimination.

Static Hashing

Goal: store and retrieve tuples with *expected* $O(1)$ I/O operations.

Design parameters:

- B : number of *buckets*.
- $h()$: hash function that maps tuples into a number $0 \leq i < B$.

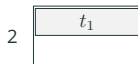
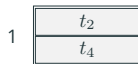
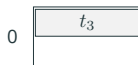
Example: $B = 4$ buckets.

- $h(t_1) = 2$

- $h(t_2) = 1$

- $h(t_3) = 0$

- $h(t_4) = 1$



Good Hash Functions¹⁴?

A **perfect** hash function has the following properties:

- (1) it is **fast** to compute
- (2) it avoids *collisions*: if $t_1 \neq t_2$ then $h(t_1) \neq h(t_2)$
- (3) it avoids *clustering*: hash values are “nicely” spread in $[0, B)$

A good (and usable) hash function *approximates* these properties.

Typical hash function $\boxed{h(t) = f(t) \bmod B}$, where $f(t)$ computes a number out of the tuple (e.g., sum of first 20 bytes in the key).

¹⁴https://en.wikipedia.org/wiki/Hash_function

Insertions with Static Hash Files

Every new tuple t_i is stored on block $h(t_i)$.

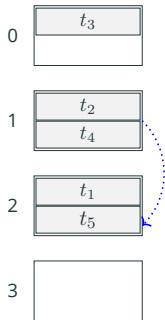
If block $h(t_i)$ has room for another tuple, just make the insertion.

If $h(t_i)$ is full, there are two options:

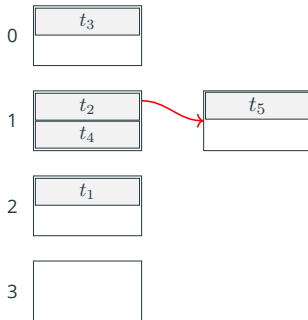
- (1) Probe subsequent buckets **sequentially** until an empty slot is found.
- (2) Grow the buckets with *overflow blocks*.

Example: insert new tuple t_5 with $h(t_5) = 1$

Sequential probing:



Using overflow blocks:



Sequential probing complicates deletions and searches.

Assume we further insert t_6 with $h(t_6) = 2$.

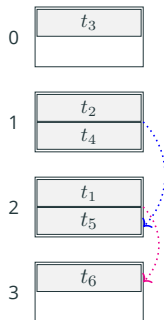
What to do if we wish to **delete** t_5 ?

Option 1: *compact* the file

- Scan the file until the first empty slot after t_5 .
- Re-hash all tuples out of place in that interval.

Option 2: put a “tombstone” marker ()

- **Ignore tombstones** when **searching**
- Re-use the spot if possible



Overflow blocks complicate deletions and searches.

Assume we further insert t_5, t_6 and t_7 with $h(t_5) = h(t_6) = h(t_7) = 1$.

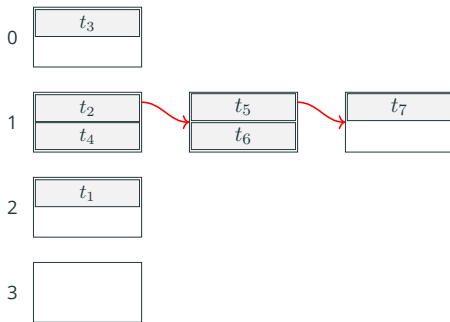
What to do if we wish to **delete** t_5 ?

Option 1: *compact* the file

- May require moving tuples through many blocks

Option 2: leave gaps

- Reduces I/O efficiency when searching



Best case scenario (one block per bucket and no collisions): the cost of all operations (insertion, search, and deletion) is 1 I/O operation.

The analysis of the **average** case cost (with some collisions) gives the same cost for both probing or using overflow chains: $O(\log N)$ for all operations, where N is the number of records in the file.

The same is true for hash file in memory!

Static hashing does not scale

Static hashing works reasonably well as long as a fraction of blocks that are used remains relatively low (usually, below 75%).

If the file gets “too full” we need to:

- (1) Add more buckets (increasing B)
- (2) **Re-hash every tuple!** (because B increased!)

Know your programming language

In-memory implementations of hash sets and dictionaries/maps in Python and Java use static hashing that grow by **doubling** in size whenever they are “too full”.

Dynamic Hashing

Static hashing is bad because growing the hash file is very expensive... requiring to re-hash everything again after certain updates, which can take a very long time.

Dynamic hashing schemes aim to **grow the hash file incrementally**.

Extensible Hashing keeps a dynamic in-memory directory with pointers to disk blocks. (See 11.7 in Silberschatz et al. or 11.3.5 in Garcia-Molina et al.)

Linear Hashing grows the hash file one block at a time.

Linear Hashing

Four ingredients:

(1) Distinguish between *virtual* and *materialized* buckets.

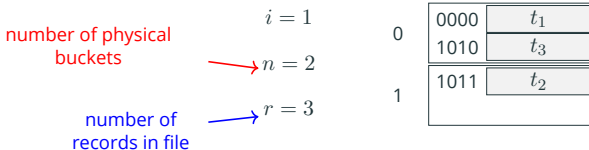
(2) Use the **last** i bits of the hash value as the bucket number.

If a tuple hashes to a virtual bucket, use the last $i - 1$ bits instead (which is guaranteed to map to a materialized bucket).

(3) Buckets can span multiple disk blocks (using overflow chains).

(4) Add more materialized buckets once the load factor goes over a threshold.

Example: $k = 4$ and $i = 1$ (meaning, two logical blocks).



Invariants: $i = \lceil \log_2 n \rceil$ and $r/n \leq C \cdot |\text{block}|$

C is the **load factor** above which the file must grow.

In the example, $|\text{block}| = 2$ and the file must grow if $r/n > 1.7$ for a maximum load factor of $C = 85\%$.

Finding the bucket of a tuple

The *logical bucket* of tuple t_i is the integer m corresponding to the lowest i bits in $h(t_i)$.

(1) $m < n$: the logical bucket is materialized.

In this case, the **bucket** of t_i is m itself.

(2) $m \geq n$: the logical bucket is virtual (i.e., it is not on disk yet).

In this case, the **bucket** of t_i is the one corresponding to the lowest $i - 1$ bits in $h(t_i)$.

Invariant

Every tuple has a **bucket** where it gets stored, even if we do not use all i bits of the hash value.

Insertions With Linear Hashing

Algorithm for inserting tuple t_i into a Linear Hash file

- (1) determine the **bucket** of t_i (slide 99)
- (2) if the bucket has room, add t_i ; otherwise, **grow the bucket using an overflow block** and add t_i
- (3) **increase r**
- (4) **if $r/n > C \cdot |\text{block}|$** , add one more bucket to the file and increase n
if $\lceil \log_2 n \rceil > i$, increase i and **re-hash** the tuples in physical block $n - 2^i$

Example: inserting t_4 with $h(t_4) = 0101$

$i = 1$			
	0	0000	t_1
$n = 2$		1010	t_3
	1	1011	t_2
$r = 3$			

Example: inserting t_4 with $h(t_4) = 0101$

$r/n = 2 > 1.7$	$i = 1$	0	0000	t_1
	$n = 2$		1010	t_3
	$r = 4$	1	1011	t_2
			0101	t_4

With the insertion, the load factor is too high and the file needs to grow (next slide).

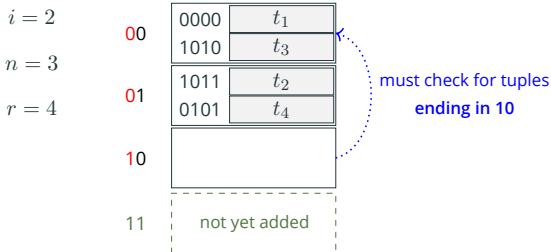
Growing the file; i Increases

Adding a bucket makes $n = 3$; since $\lceil \log_2 3 \rceil > 1$, i is increased, **doubling** the space of *logical bucket addresses*:

- The “lower half” of the space has the old buckets
- The new bucket becomes the **first** in the “upper half”:

$10_{i-1} \cdots 0_0$.

- We must check for tuples “out of place” in $00_{i-1} \cdots 0_0$



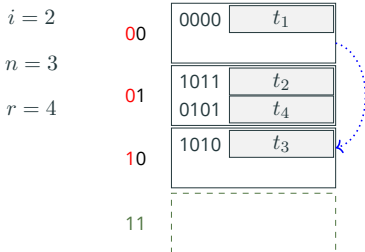
Growing the file; i Increases

Adding a bucket makes $n = 3$; since $\lceil \log_2 3 \rceil > 1$, i is increased, **doubling** the space of *logical bucket addresses*:

- The “lower half” of the space has the old buckets
- The new bucket becomes the **first** in the “upper half”:

$10_{i-1} \cdots 0_0$.

- We must check for tuples “out of place” in $00_{i-1} \cdots 0_0$



Re-hashing tuples out of place

Whenever we add a new materialized bucket (even when i remains unchanged), we must check for tuples out of place and move them accordingly.

Let the new materialized bucket be at binary address $1b_{i-1} \cdots b_0$.

Note that, prior to adding this block, any tuple with that exact hash value would have been inserted using less than i bits!

Those tuples would be in the bucket with address $0b_{i-1} \cdots b_0$.

When adding an **overflow** block; n and i stay the same

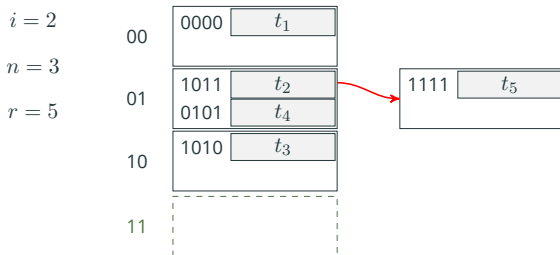
Example: adding tuple t_5 with $h(t_5) = 1111$

$i = 2$	00	0000	t_1
$n = 3$	01	1011	t_2
$r = 4$		0101	t_4
	10	1010	t_3
	11	not yet added	

n **does not increase** when adding an overflow block

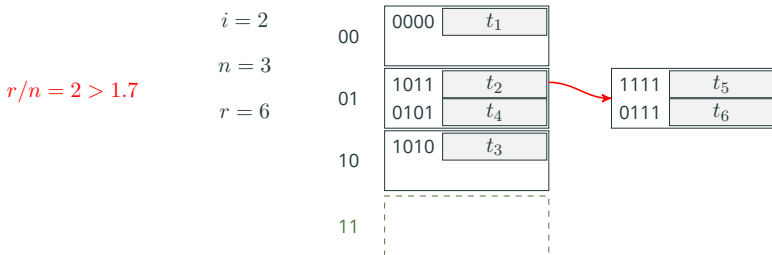
When adding an **overflow** block; n and i stay the same

Example: adding tuple t_5 with $h(t_5) = 1111$



n **does not increase** when adding an overflow block

Another example of inserting into empty slots: add tuple t_6 with $h(t_6) = 0111$



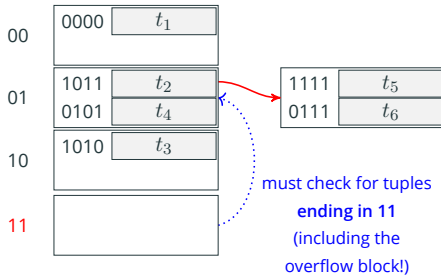
With the insertion, the load factor is too high and the file needs to grow (next slide).

Growing the file; i Stays the Same

$$i = 2$$

$$n = 3$$

$$r = 6$$

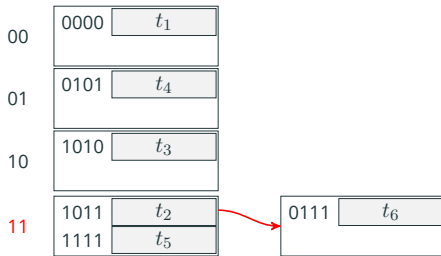


Growing the file; i Stays the Same

$$i = 2$$

$$n = 4$$

$$r = 6$$




With one more insertion we'd have $r = 7$ and $r/n > 1.7$, forcing the file to grow again (n becomes 5 and $i = \lceil \log_2 5 \rceil = 3$, doubling the number of bucket addresses as in Case 1 as in slide 102).

Linear Hashing: Summary

Inserting duplicates? No special treatment... just add the new tuples wherever they hash to.

Searching? Follow steps (1)–(3) of the insertion algorithm (slide 99).

Deletions? Just put a “tombstone” mark () instead of reversing the insertion algorithm.

Access costs?

- searches, insertions and deletions require, **in expectation**, at most two I/O operations (1 read and 1 write, if the file grows).

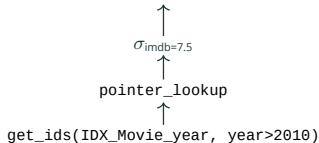
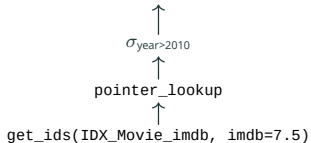
Multi-key Access Methods

Multi-Attribute Access

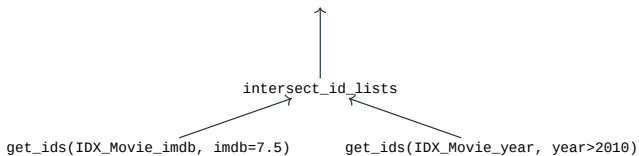
Goal: finding records satisfying multiple selection criteria at once

- Ex: movies where $\text{year} > 2010$ AND $\text{imdb} = 7.5$.

Option I: use an index on **either** attribute, and verify the other predicate on the other attribute for each tuple.



Option II: use two indexes, each on a different attribute, and **merge** the lists of tuple ids.



Option III: use a **multidimensional** index.

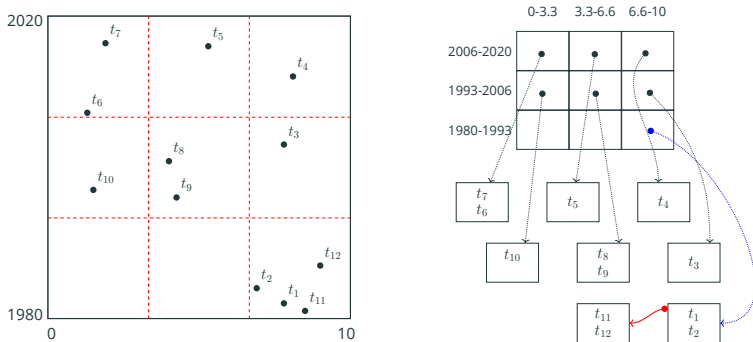
As the name suggests, a multidimensional index considers multiple attributes (the dimensions) at the same time.

Many of them were designed to deal with numeric data, but they can work with all kinds of attributes.

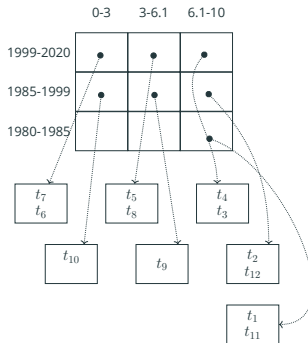
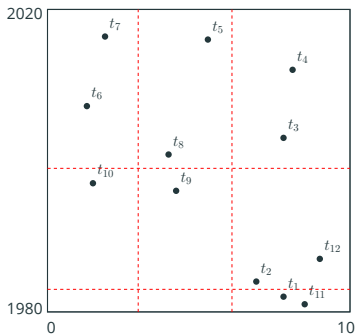
In the examples that follow, we will consider multidimensional indexes on (title, year) keys.

Grid files are a *hashing-based* representation of multiple points in a n -dimensional space.

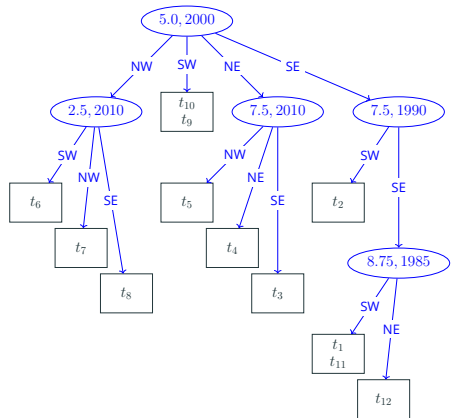
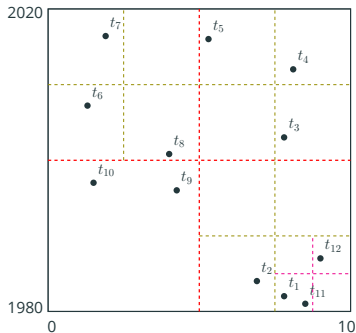
Each cell of the grid points to (a chain) of **bucket files**, with pointers to actual tuples (in the data file) that fall in that cell.



The boundaries of the grid **need not** be uniformly spaced across dimensions. Moving boundaries to “balance” the number of points per grid cell is an interesting optimization problem.



A **Quadtree** divides the space into four quadrants, recursively as needed, until each quadrant corresponds to a single block (of a bucket file).¹⁵



¹⁵<https://en.wikipedia.org/wiki/Quadtree>

All multidimensional indexes extend to more than two dimensions. However, as the number of dimensions grow, so does the complexity of the index **and** the number of empty regions that need to be represented.

The costs of creating and updating these indexes depends on many design decisions of their creators, but in general they required more work than updating unidimensional B+trees or hash files.

One very important multidimensional index (which is covered in the labs), designed for spatial data (e.g., maps), is the “region tree” (**R tree**). Like with B+trees, nodes in the R tree correspond to disk blocks and represent (MBRs of) sub-regions of the space, instead of individual points.

Other Aspects

Tuple/Row identifiers

Most DBMSs assign *identifiers* which are unique to every tuple in the same table,¹⁶ called *tuple* ids or *row* ids.

For example, in SQLite, every tuple is assigned a 64-bit signed integer key that uniquely identifies the tuple in the table—you insert the same data twice, you get two different tuples with different ids.

Tuple ids are typically visible to the application.

In SQLite, one can even update them.

```
sqlite> CREATE TABLE Temp(name CHAR(20));
sqlite> INSERT INTO Temp
...> VALUES ('Susan'), ('Bob'), ('Susan');
sqlite> SELECT rowid, * FROM Temp;
1|Susan
2|Bob
3|Susan
sqlite>
```

¹⁶Sometimes the identifiers are unique within the whole database.

Tuple ids were initially introduced in relational systems to support the storage of objects in object-oriented programming languages, where object identity is important. Two objects are considered different if their ids are different, regardless of values.

Tuple ids can be useful in some cases, such as eliminating duplicate tuples in relations without keys.

In some DBMSs, tuple ids are actually symbolic references to the record

for the tuple inside the corresponding data file.

One interesting use of tuple ids is in implementing column stores.

```
sqlite> DELETE FROM Temp WHERE rowid IN (  
...>     SELECT rowid FROM Temp t  
...>     WHERE EXISTS (SELECT t2.rowid  
...>                     FROM Temp t2  
...>                     WHERE t2.name=t.name  
...>                     AND t2.rowid<t.rowid));  
sqlite> SELECT rowid, * FROM Temp;  
1|Susan  
2|Bob  
sqlite>
```

Column-oriented stores

With a tuple-oriented store (slide 8) each record corresponds to an entire tuple of a table. For some applications, this design turns out to be sub-optimal.

In **scientific applications** it is common for some relations to have hundreds or thousands of attributes, leading to records that are quite large, and a low record/block ratio.

Typical queries in these applications involve only a small fraction of the attributes in the **WHERE** clause and in the value expressions, resulting in a high I/O “per useful attribute” cost.

On Line Analytical Processing queries apply over business data often computing aggregations and groupings defined over a small fraction of the schema attributes.

The idea behind a column-store is to have separate data files for each of the attributes in the relation.

For example, the Movie relation in the running example would be stored in four separate data files like so:

id	title
1	Ghostbusters
2	Big
3	Lost in Translation
4	Wadjda
5	Ghostbusters

id	year
1	1984
2	1988
3	2003
4	2012
5	2016

id	imdb
1	7.8
2	7.3
3	7.8
4	8.1
5	5.3

id	director
1	Ivan Reitman
2	Penny Marshall
3	Sofia Coppola
4	Haifaa al-Mansour
5	Paul Feig

This design has some advantages from a storage point of view:

- records are very simple: one id and one attribute;
- **NULL** values are easy to deal with: just ignore them;
- **data compression** can be used in each file separately.

What about query processing?

id	title
1	Ghostbusters
2	Big
3	Lost in Translation
4	Wadjda
5	Ghostbusters

id	year
1	1984
2	1988
3	2003
4	2012
5	2016

id	imdb
1	7.8
2	7.3
3	7.8
4	8.1
5	5.3

id	director
1	Ivan Reitman
2	Penny Marshall
3	Sofia Coppola
4	Haifaa al-Mansour
5	Paul Feig

It is possible to evaluate SQL quite efficiently with column stores. In fact, these systems outperform tuple-stores by quite a margin on most analytical queries.

The main idea is to process each condition in the **WHERE** clause separately, producing a list of tuple ids satisfying each condition. Then merge these lists based on the logical connector: **AND** means intersection, **OR** means union.

Is there anything this design is **bad** for?

- YES: **SELECT** * ... queries.