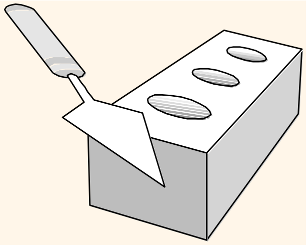
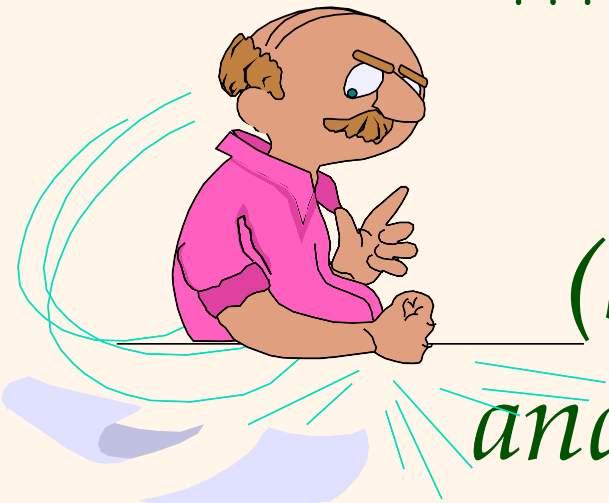


Introduction to Data Management

*** The "Flipped" Edition ***

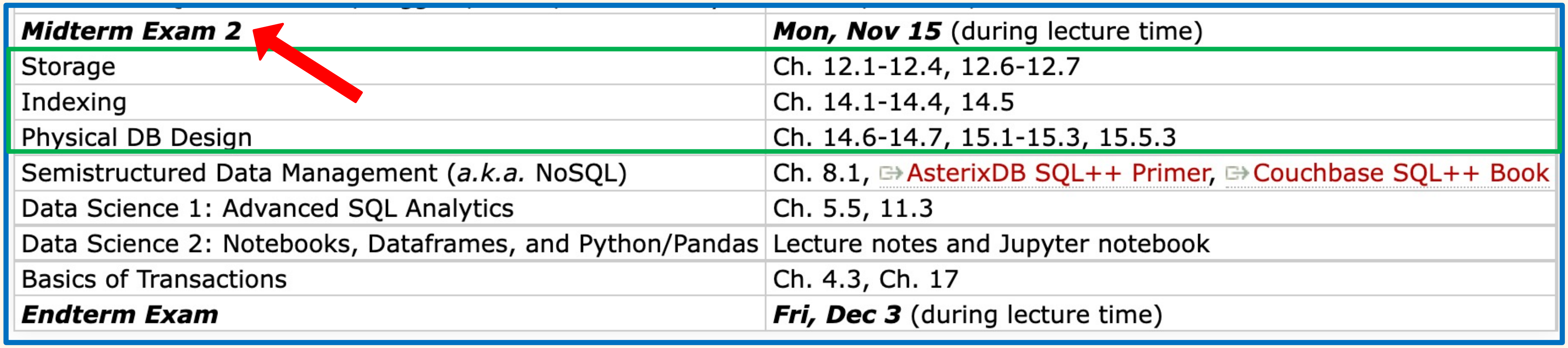


Lecture #20 (Storage & Indexing III and Physical DB Design I)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Roadmap reminder:



Midterm Exam 2	***Mon, Nov 15*** (during lecture time)
Storage	Ch. 12.1-12.4, 12.6-12.7
Indexing	Ch. 14.1-14.4, 14.5
Physical DB Design	Ch. 14.6-14.7, 15.1-15.3, 15.5.3
Semistructured Data Management (*a.k.a.* NoSQL)	Ch. 8.1, ⇨AsterixDB SQL++ Primer, ⇨Couchbase SQL++ Book
Data Science 1: Advanced SQL Analytics	Ch. 5.5, 11.3
Data Science 2: Notebooks, Dataframes, and Python/Pandas	Lecture notes and Jupyter notebook
Basics of Transactions	Ch. 4.3, Ch. 17
Endterm Exam	***Fri, Dec 3*** (during lecture time)

- HW #6 should be wrapping up now!
  - Second in the series of ***SQL-based*** HW assignments
  - Due **this Friday** @ **6 PM** (w/usual 24-hour late window)
- Midterm #2 is now one weekend away *(wow!)*
  - Monday (Nov. 15), conducted just like Midterm #1
  - In person, Gradescope + hard copy cheat sheet, assigned seats
- **Today:** Finish Storage & Indexing, then move to Physical DB Design!

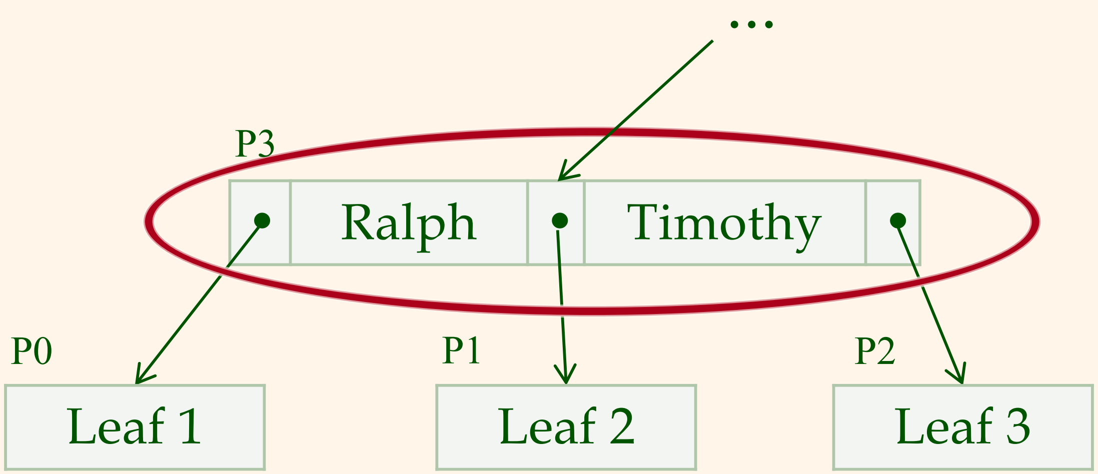
A Note on B+ Tree "Order"

- (Mythical!) *order* (**d**) concept replaced by physical space criterion in practice (***"at least half-full*"**)**.**
  - Index pages can typically hold many more entries than leaf pages.
  - Variable-sized records and search keys mean that different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries in the tree's leaf pages.

(Page Implementation Details)

Q: What if you were to "open up" a *B+ Tree* ***page***?

- Control info (e.g., level, # children, free space offset)
- Search key array (with possible on-page indirection for variable-length data, using offsets), or a key/data array – for non-leaf *vs.* leaf pages, respectively
- Child pointer array, where pointer = page id on disk!

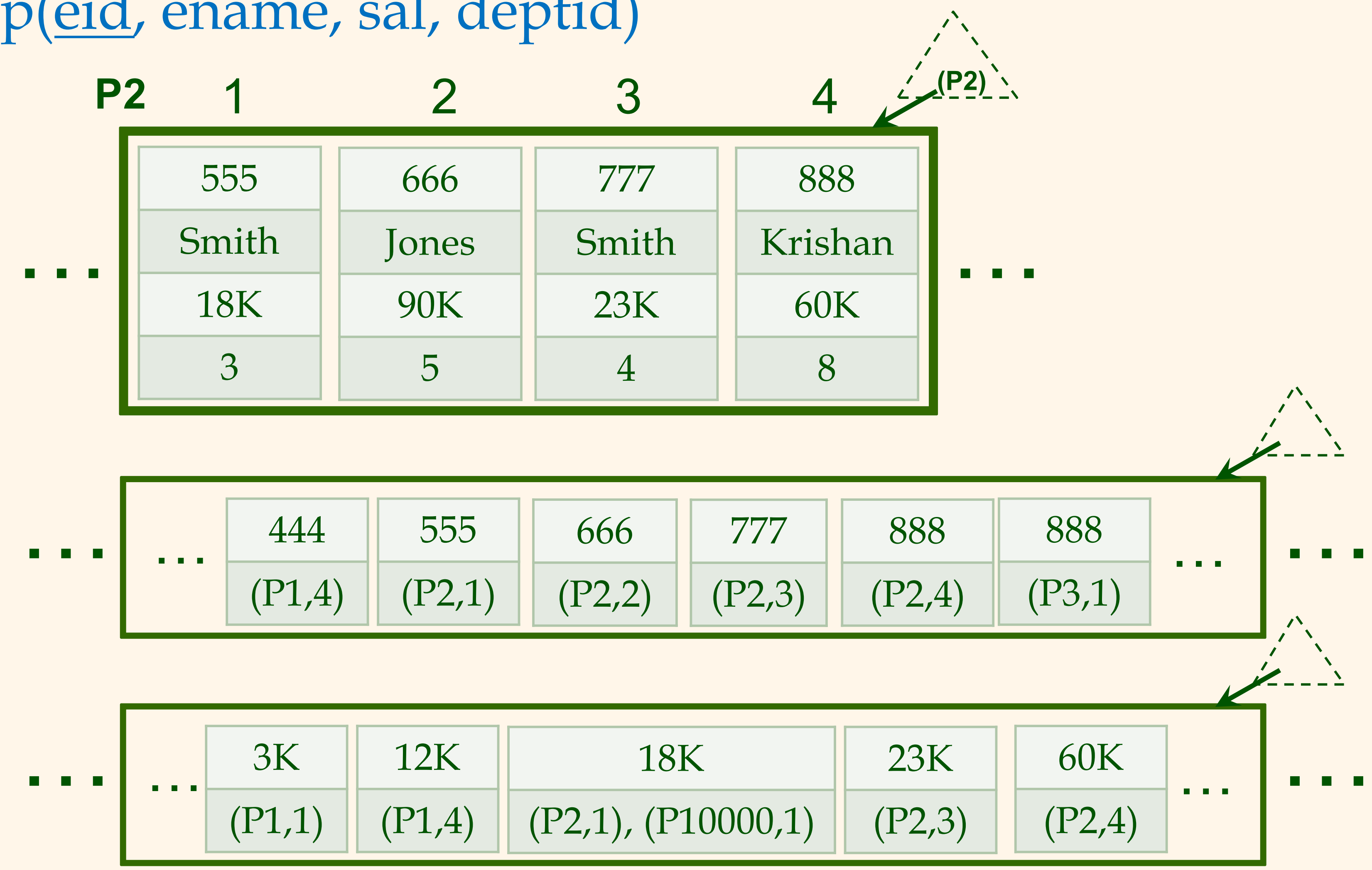


0	Level (1)
	NumChildren (3)
8	Free offset (40)
	Key 0 offset (32)
	Key 1 offset (37)
20	Child 1 page id (P0)
	Child 2 page id (P1)
	Child 3 page id (P2)
32	Key 0 ("Ralph")
37	Key 1 ("Timothy")
40	...

} *not to scale...*

(Leaf Page I(k) Alternatives Revisited)

Ex: Emp(eid, ename, sal, deptid)



Alternative 1:
(records)

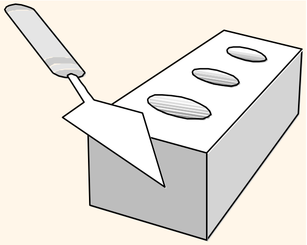
P2	1	2	3	4
…	555	666	777	888	…
	Smith	Jones	Smith	Krishan
	18K	90K	23K	60K
	3	5	4	8

Alternative 2:
(RIDs)

…	…	444	555	666	777	888	888	…	…
		(P1,4)	(P2,1)	(P2,2)	(P2,3)	(P2,4)	(P3,1)		

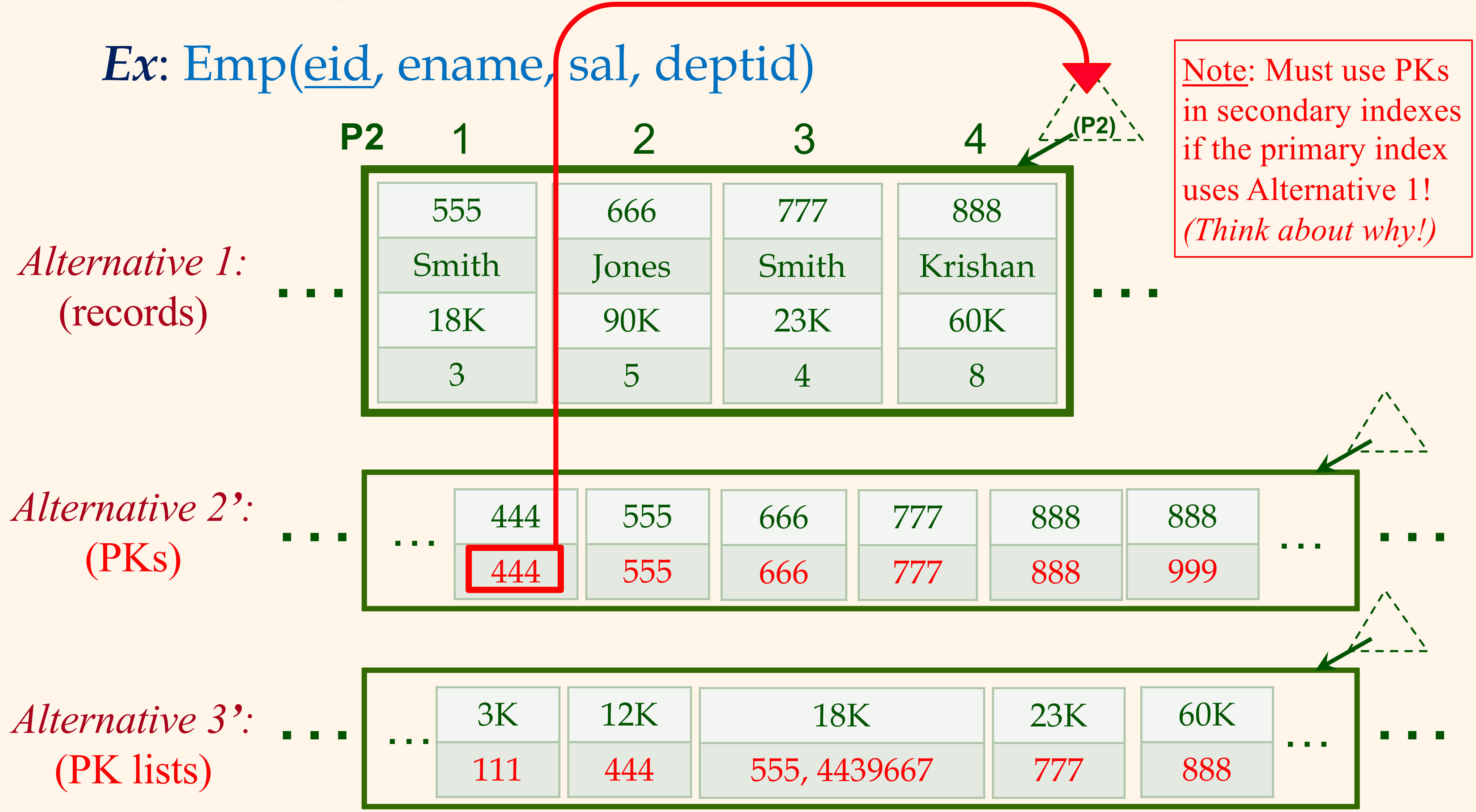
Alternative 3:
(RID lists)

…	…	3K	12K	18K	23K	60K	…	…
		(P1,1)	(P1,4)	(P2,1), (P10000,1)	(P2,3)	(P2,4)		



(Leaf Page I(k) Alternatives, cont.)

Ex: Emp(eid, ename, sal, deptid)



Alternative 1: (records)

P2 1	2	3	4
555	666	777	888
Smith	Jones	Smith	Krishan
18K	90K	23K	60K
3	5	4	8

Alternative 2': (PKs)

…						…
	444	555	666	777	888	888
	444	555	666	777	888	999

Alternative 3': (PK lists)

…					…
3K	12K	18K	23K	60K	
111	444	555, 4439667	777	888	

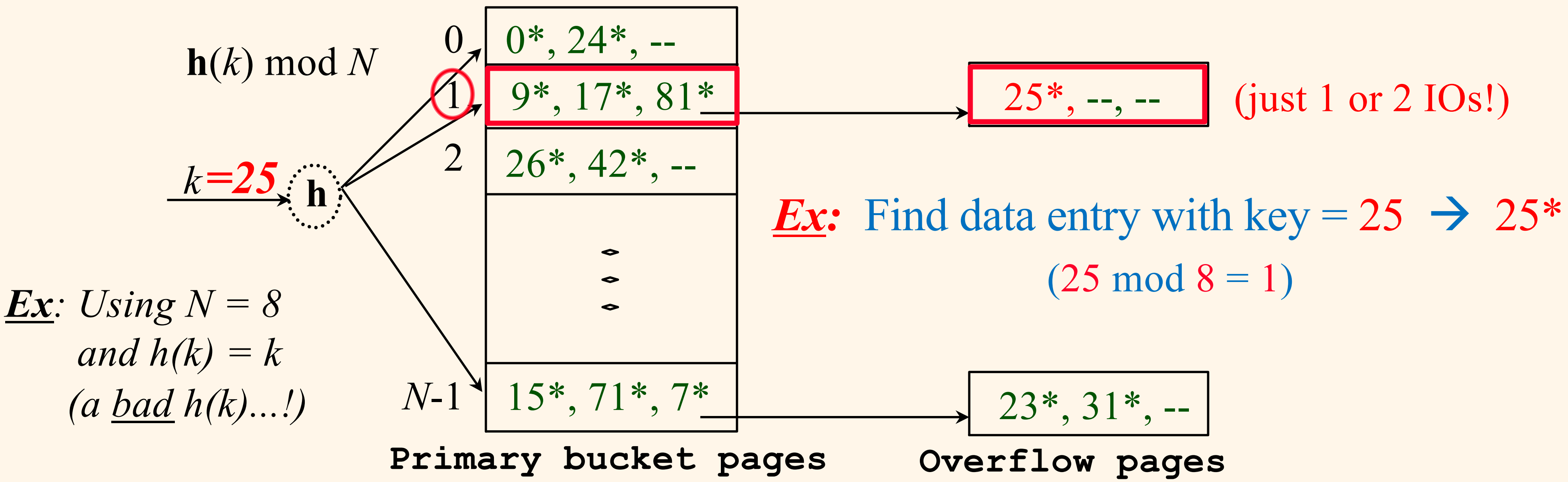
Note: Must use PKs in secondary indexes if the primary index uses Alternative 1! *(Think about why!)*

Hash-Based *Indexes*

- *Hash-based* indexes are fast for *equality selections*. ***Cannot*** support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ***ISAM*** vs. B+ trees.
- *As for any index, 3 alternatives for data entries* **k***:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
  - Choice is orthogonal to the *indexing technique!*

Static Hashed Indexes

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod *N* = bucket (page) to which data entry with key *k* belongs. (*N* = # of buckets)



h(*k*) mod *N*

k=**25**

h

0 — 0*, 24*, --

1 — 9*, 17*, 81* → 25*, --, -- (just 1 or 2 IOs!)

2 — 26*, 42*, --

N-1 — 15*, 71*, 7* → 23*, 31*, --

Primary bucket pages

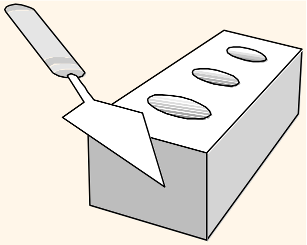
Overflow pages

Ex: Find data entry with key = 25 → 25*

(25 mod 8 = 1)

Ex: *Using N = 8*
and h(k) = k
(a bad h(k)…!)

Static Hashed Indexes (Cont'd.)

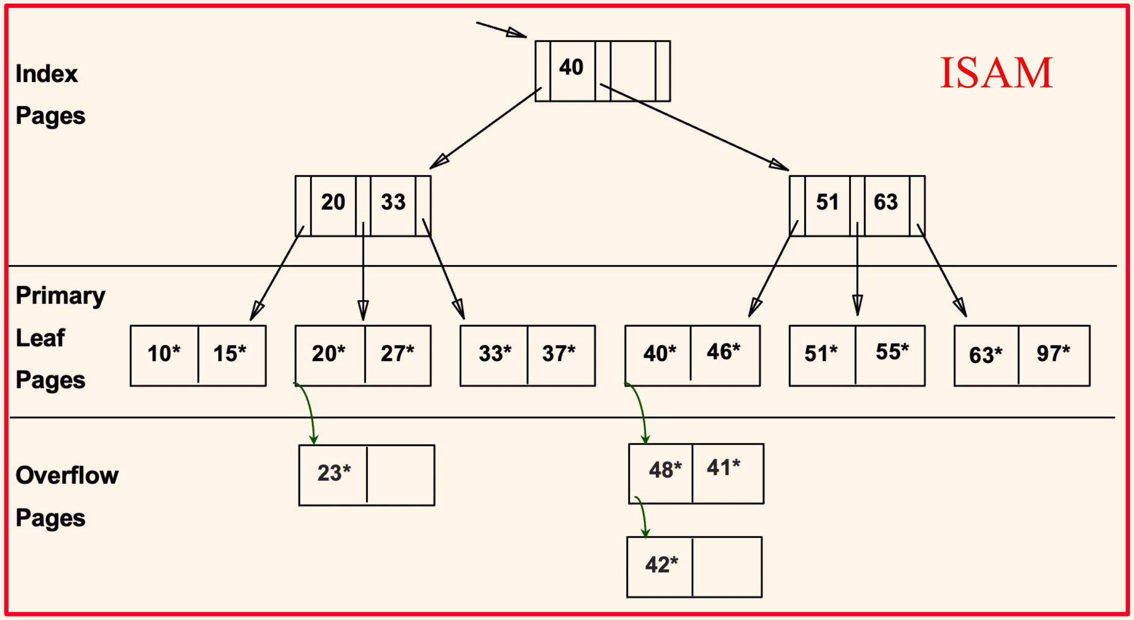


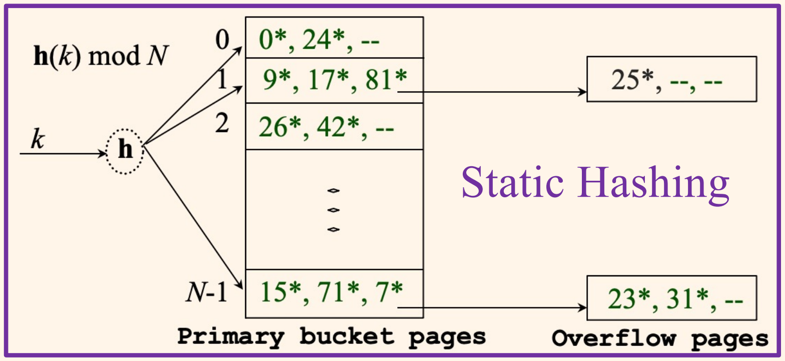
- Buckets contain *data entries* (like for ISAM or B+ trees) – very similar to what we just looked at.
- Hash function works on *search key* field of record *r*. Must distribute values over range *0...N-1*.
  - ***h***(*key*) = (a * *key* + b) *mod N* works fairly well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance. (Analogous to ISAM.)
  - *Extendible Hashing* and *Linear Hashing*: More dynamic approaches that address this problem. (Take CS122c!)

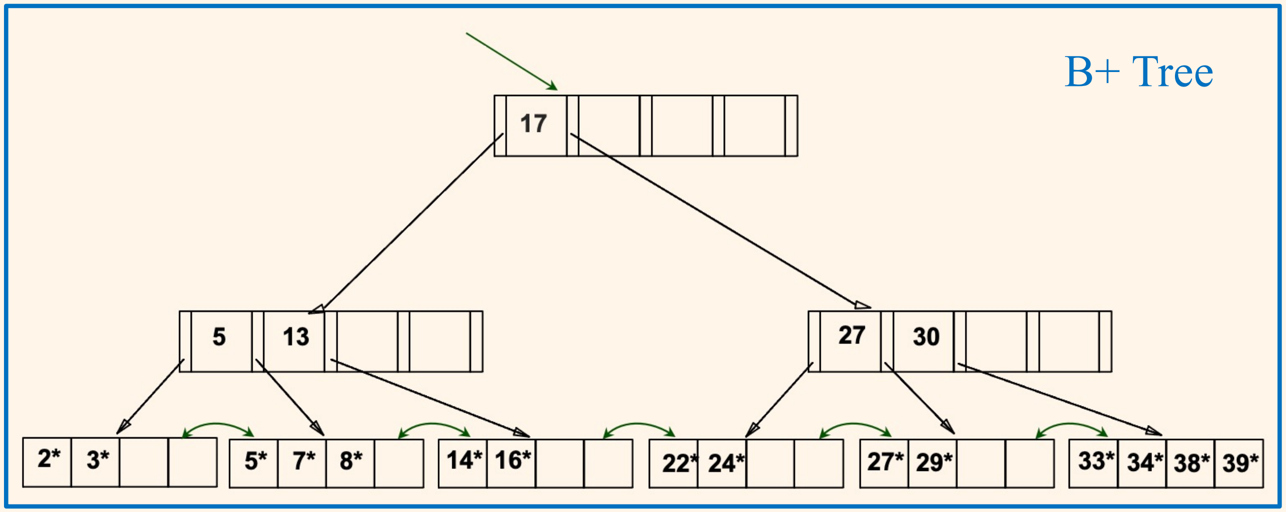
Indexing Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure. (Prehistoric B+ Tree!)
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure. (Widely used!)
  - Inserts/deletes leave tree height-balanced; $log_F N$ cost.
  - High fanout ***F*** → tree depth rarely more than 3-4.
- Hashed indexes are an option for equality searches.

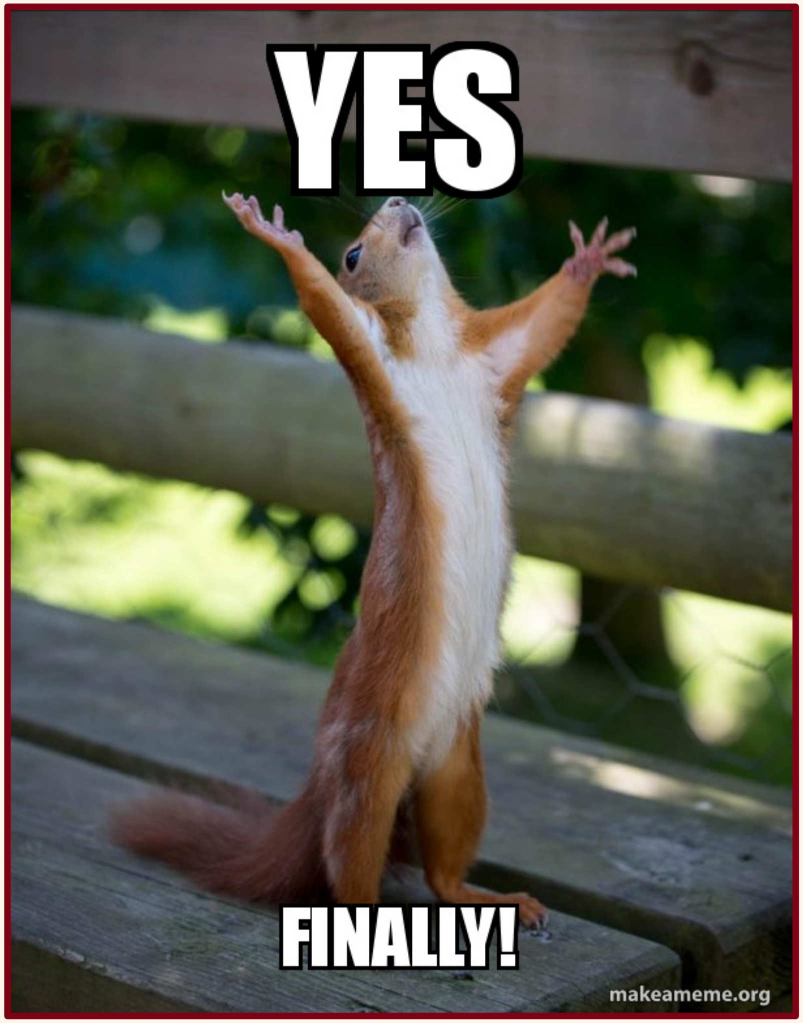
Indexing Summary (cont.)

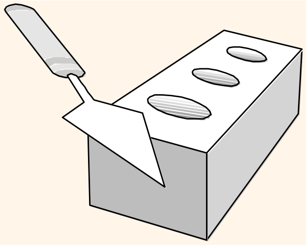






Next*: *Physical Database Design





Physical DB Design: Overview

- After ER design, schema refinement, and the definition of any views, we have the *conceptual* and *external* schemas for our database.
- Next step is to *choose indexes*, make *clustering decisions*, and *refine* the conceptual and external *schemas* (if needed) to meet *performance goals*.
- Start by ***understanding the workload***:
  1. Most important queries and how often they arise.
  2. Most important updates and how often they arise.
  3. Desired performance goals for those queries/updates?

Decisions to Be Made Include...

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be their search keys? Should we build several indexes?
- For each index, what kind of an index should it be?
  - B+ tree? Hashed? Clustered? Unclustered?
- Should we make changes to the conceptual schema?
  - Consider alternative normalized schemas? (There are multiple choices when decomposing into BCNF, etc.)
  - Should we "undo" some decomposition steps and settle for a lower normal form? (*"Denormalization."*)
  - Horizontal partitioning, materialized views, replication, ...

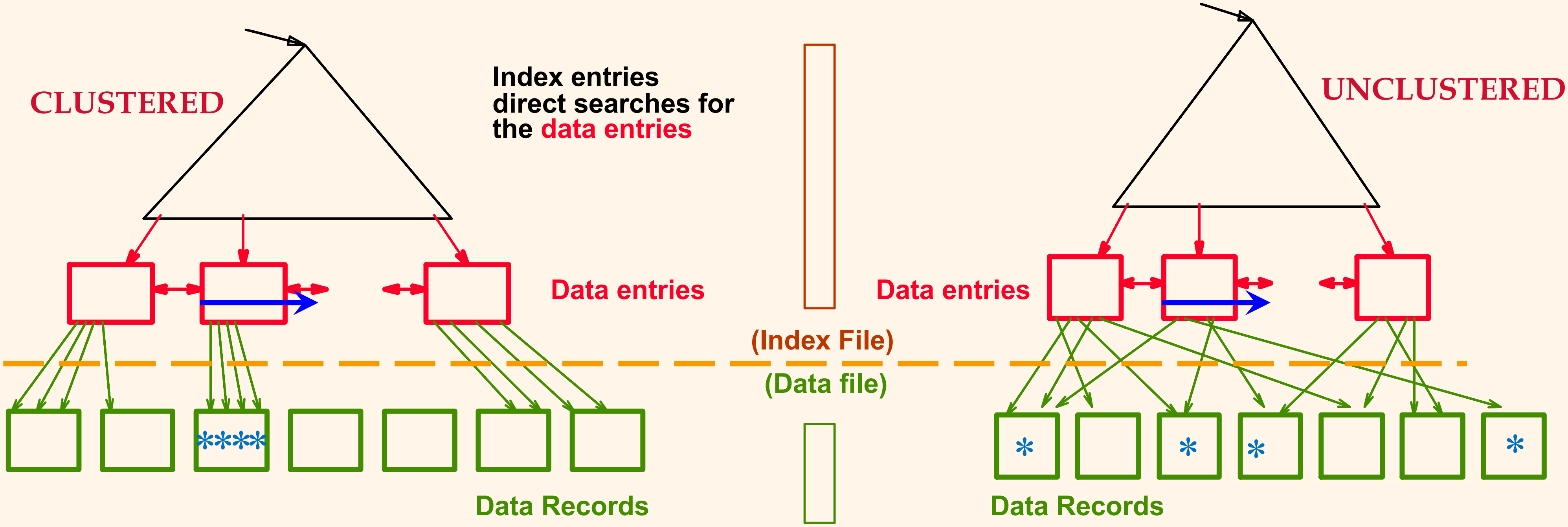
Understanding the Workload

- For each **query** in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes appear in selection/join conditions? (And ***how selective*** are those conditions expected to be?)
- For each **update** in the workload:
  - Which attributes are involved in selection/join conditions? (And ***how selective*** are those conditions likely to be?)
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

*Index Classification (**Review**)*

- *Primary* vs. *secondary*: If index search key contains the primary key, this is called the *primary* index.
  - *Unique* index: Search key contains a *candidate* key.
- *Clustered* vs. *unclustered*: If the order of data entries is the same as, or nearly so, the order of stored data records, we have a clustered index.
  - A table can be clustered on *at most one* search key.
  - Cost of retrieving data records via an index varies *greatly* based on whether index is clustered or not!
  - Some systems always cluster on the primary key.

Clustered vs. Unclustered Indexes ***(Reminder)***



CLUSTERED

Index entries
direct searches for
the data entries

Data entries

(Index File)

(Data file)

Data Records

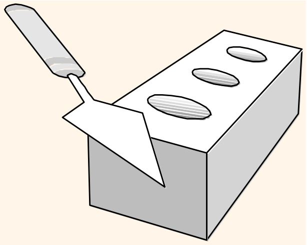
UNCLUSTERED

Data entries

Data Records

(Read each page once.)

*(Read more pages – and **repeatedly**!)*



Choice of Indexes (Cont'd.)

- One approach: Consider the most important queries in turn. Consider the best query plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - This means we must understand and see how a DBMS evaluates its queries. (*Query execution plans*.)
  - Let's start by discussing simple 1-table queries!
- Before creating an index, must also consider its impact on updates in the workload.
  - *Trade-off*: Indexes can make queries go faster, but updates will become slower. (Indexes require disk space, too.)

Index Selection Guidelines

- Attributes in **WHERE** clause are candidates for index keys.
  - Exact match condition → hashed index (or B+ tree).
  - Range query → B+ tree index.
    - Clustering especially useful for range queries, but can also help with *equality queries with* ***duplicate values*** (i.e., a non-key field index).
- **Multi-attribute** search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes in key matters for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries (e.g., aggregates / grouped aggregates).
    - *Note*: For index-only strategies, data clustering isn't important!
- Choose indexes that benefit **as many queries** as possible.
  - Only **one** index can be clustered per relation, so choose it based on important queries that can benefit the most from clustering.

Some Clustered Index Use Cases

```
SELECT E.dno
FROM   Emp E
WHERE E.age > 40;
```

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Should the index be clustered?

```
SELECT E.dno,
          COUNT (*)
FROM   Emp E
WHERE E.age > 10
GROUP BY E.dno;
```

- Consider the GROUP BY query.
  - If most tuples have *E.age* > 10, using *E.age* index and grouping the retrieved tuples may be costly.
  - Clustered *E.dno* index may win!

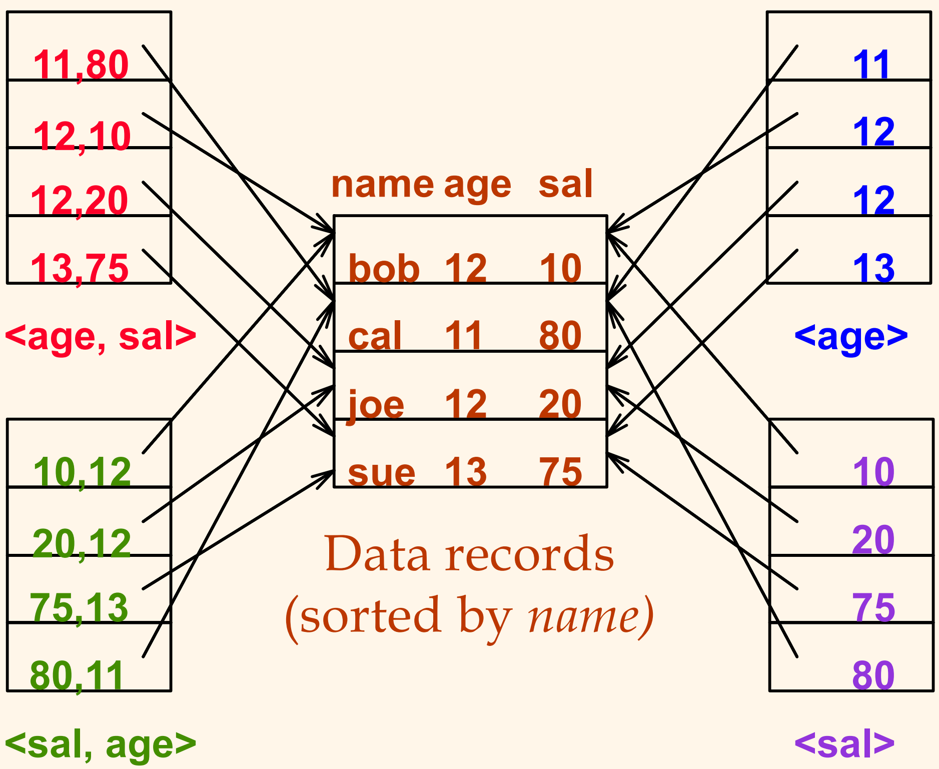
```
SELECT E.dno
FROM   Emp E
WHERE E.hobby='Stamps';
```

- Equality queries & duplicates:
  - Clustering on *E.hobby* helps!

Indexes with Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - (age=20 *AND* sal=75)
  - Range query: Some field value is a range, not a constant. E.g. again wrt <sal,age> index:
    - age=20; or (age=20 *AND* sal > 10)
- Data entries in index sorted by search key to support such range queries.
  - ***Lexicographic order***

Various composite key indexes using lexicographic (ASC) order.



<age, sal>: 11,80 | 12,10 | 12,20 | 13,75

<sal, age>: 10,12 | 20,12 | 75,13 | 80,11

name	age	sal
bob	12	10
cal	11	80
joe	12	20
sue	13	75

11 12 12 13 <age>

10 20 75 80 <sal>

Data records (sorted by *name*)

Data entries in index sorted by <*sal,age*>

Data entries sorted by <*sal*>

Composite Search Keys

- To retrieve Emp records with *age*=30 **AND** *sal*=4000, an index on <*age,sal*> or <*sal,age*> would be better than an index only on *age* or an index only on *sal*.
    - *Note*: Choice of index key is orthogonal to clustering.
- If condition is: 20<*age*<30 **AND** 3000<*sal*<5000:
    - Clustered B+ tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 **AND** 3000<*sal*<5000:
    - Clustered <*age,sal*> index ***much*** better than <*sal,age*> index! (***Think about why:*** Draw a picture of the index!)
- Composite indexes are larger; updated more often.

Index-Only Query Plans

- Some queries can be answered without retrieving ***any*** tuples from one or more of the relations involved if a suitable index is available.

<E.dno>

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno;
```

<E.dno,E.sal>
B+ tree index!

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno;
```

<E. age,E.sal>
B+ tree index!

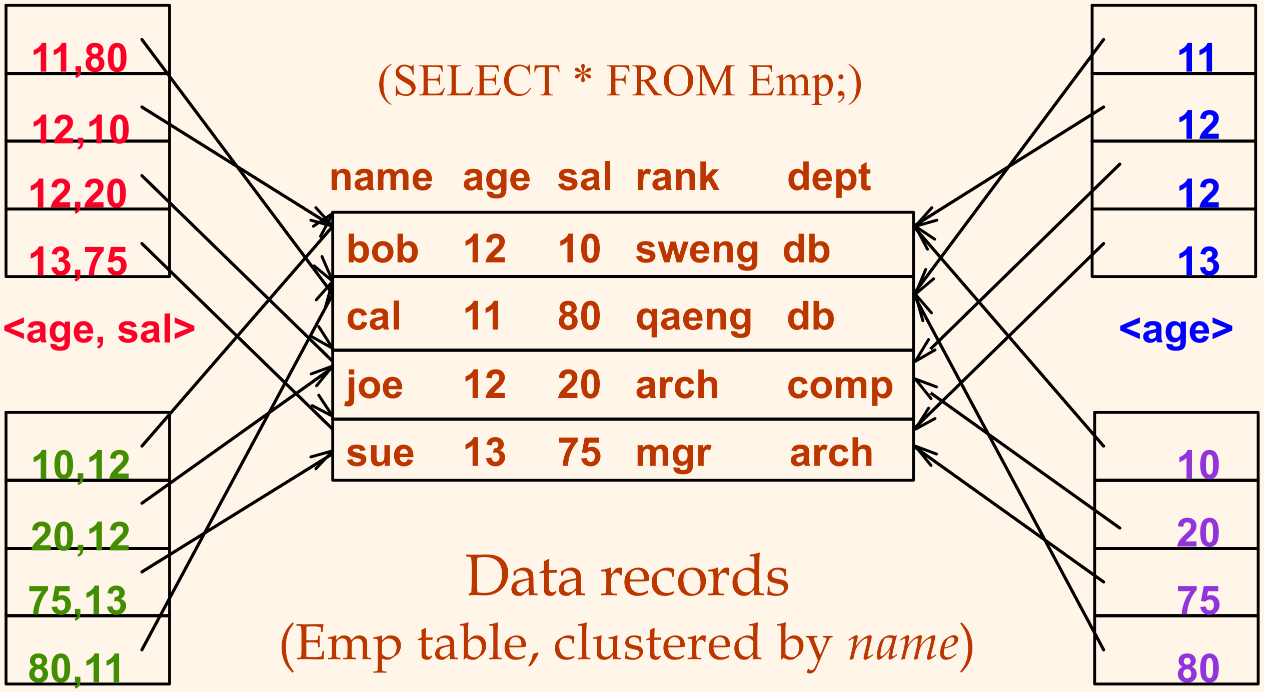
```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000;
```

*(Sometimes called a "**covering index**" for the given query.)*

Some Illustrated Index-Only Plans

```
SELECT sal, age
FROM Emp
WHERE age > 11
  AND age <= 13;
```

```
SELECT age,
   COUNT(*)
FROM Emp
GROUP BY age;
```



11,80
12,10
12,20
13,75

<age, sal>

10,12
20,12
75,13
80,11

<sal, age>

(SELECT * FROM Emp;)

name	age	sal	rank	dept
bob	12	10	sweng	db
cal	11	80	qaeng	db
joe	12	20	arch	comp
sue	13	75	mgr	arch

Data records
(Emp table, clustered by *name*)

11
12
12
13

<age>

10
20
75
80

<sal>

```
SELECT sal,
   AVG(age)
FROM Emp
GROUP BY sal;
```

```
SELECT
   DISTINCT sal
FROM Emp;
```

Note: The index files are each much smaller than the main file!

To Be Continued...