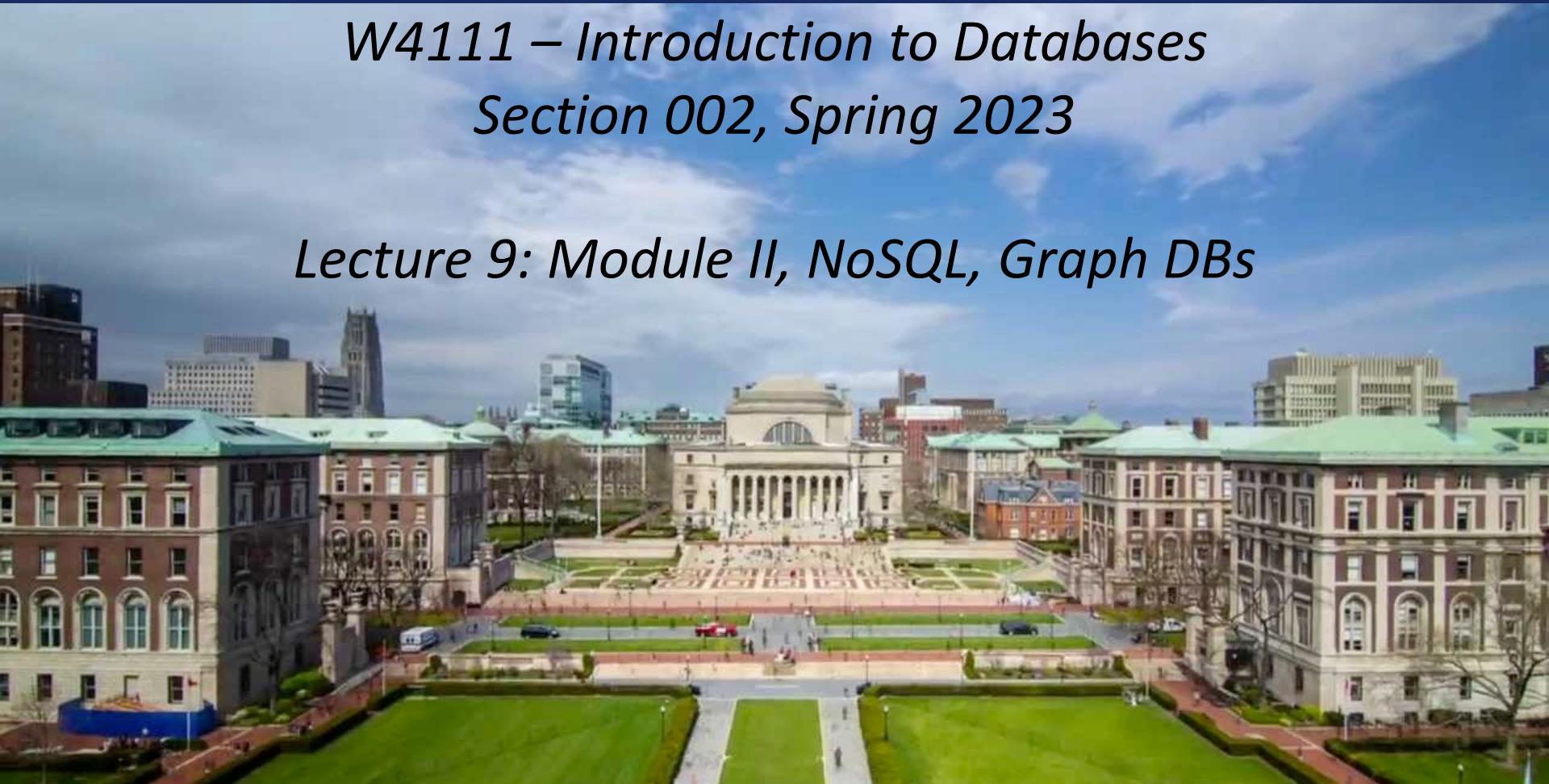


*W4111 – Introduction to Databases
Section 002, Spring 2023*

Lecture 9: Module II, NoSQL, Graph DBs



Contents

Contents

Module II Continued

Module II – DBMS Architecture and Implementation Overview and Reminder

Module II – DBMS Architecture and Implementation

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

- 
1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

Module II – DBMS Architecture and Implementation

- 
- 
2. Give users the ability to *query* the data (a “query” is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.
 3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.
 4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.
 5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Database Systems: The Complete Book (2nd Edition)

by [Hector Garcia-Molina](#) (Author), [Jeffrey D. Ullman](#) (Author), [Jennifer Widom](#) (Author)

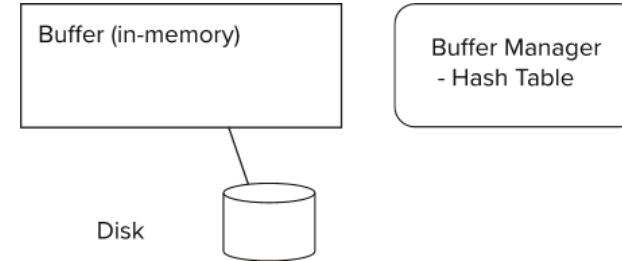
Memory Management

Buffer Management



Storage Access

- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



Architecture

- OS allocates percent of main memory to running applications.
- Allocation based on demand and workload importance.
- OS normally manages page replacement.
- OS controls the replacement for the buffer pool because of ability to predict access.

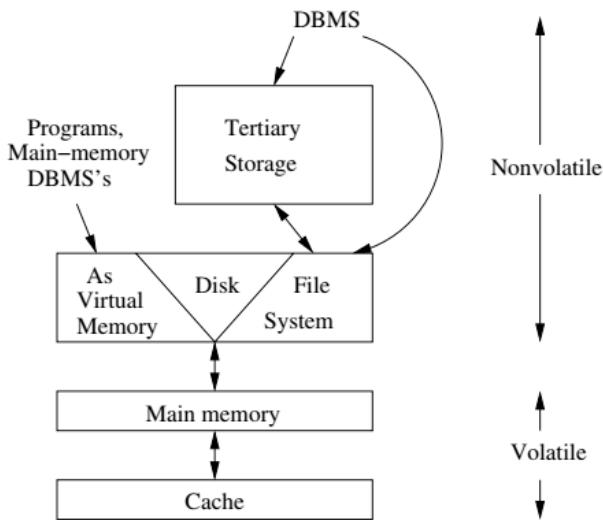
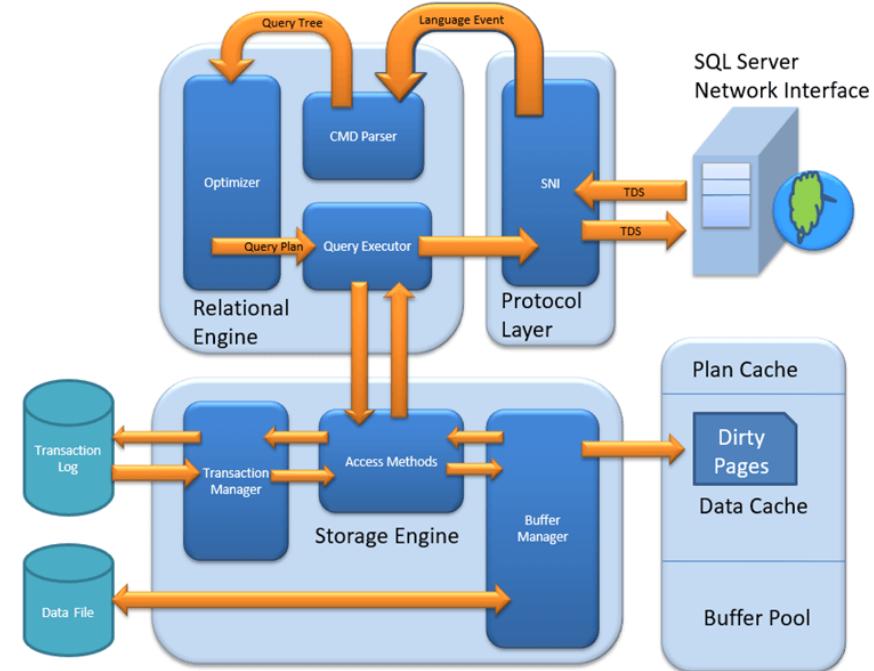
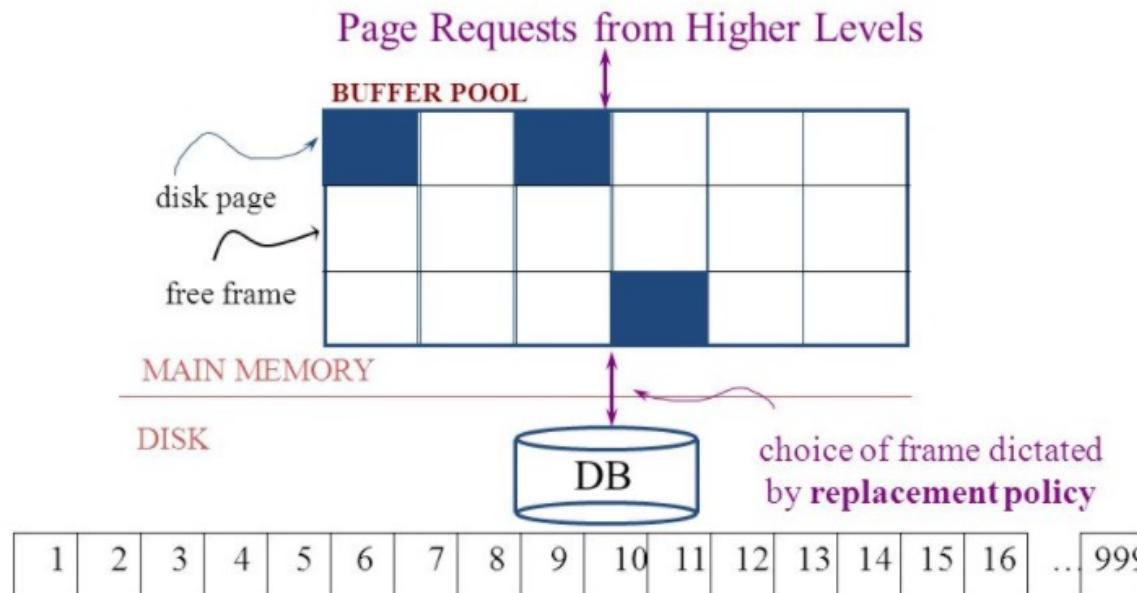


Figure 13.1: The memory hierarchy



The Logical Concept

- The DBMS and queries can only manipulate in-memory blocks and records.
- A very, very, very small fraction of all blocks fit in memory.





Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
 - If the block is already in the buffer, buffer manager returns the address of the block in main memory
 - If the block is not in the buffer, the buffer manager
 - Allocates space in the buffer for the block
 - Replacing (throwing out) some other block, if required, to make space for the new block.
 - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
 - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
 - **Pin** done before reading/writing data from a block
 - **Unpin** done when read /write is complete
 - Multiple concurrent pin/unpin operations possible
 - Keep a pin count, buffer block can be evicted only if pin count = 0
- **Shared and exclusive locks on buffer**
 - Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
 - Readers get shared lock, updates to a block require exclusive lock
 - **Locking rules:**
 - Only one process can get exclusive lock at a time
 - Shared lock cannot be concurrently with exclusive lock
 - Multiple processes may be given shared lock concurrently



Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
 - Idea behind LRU – use past pattern of block references as a predictor of future references
 - LRU can be bad for some queries
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loops

```
for each tuple  $tr$  of  $r$  do  
  for each tuple  $ts$  of  $s$  do  
    if the tuples  $tr$  and  $ts$  match ...
```



Buffer-Replacement Policies (Cont.)

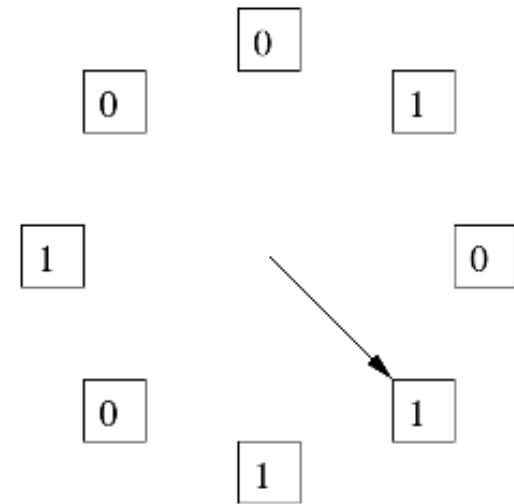
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
 - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager may reorder writes
 - Can lead to corruption of data structures on disk
 - E.g., linked list of blocks with missing block on disk
 - File systems perform consistency check to detect such situations
 - Careful ordering of writes can avoid many such problems

Replacement Policy

- The *replacement policy* is one of the most important factors in database management system implementation and configuration.
- A very simple, introductory explanation is (https://en.wikipedia.org/wiki/Cache_replacement_policies).
 - There are a lot of possible policies.
 - The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as [Bélády's optimal algorithm](#)/simply optimal replacement policy or [the clairvoyant algorithm](#).
- All implementable policies are an attempt to approximate knowledge of the future based on knowledge of the past.
- Least Recently Used is based on the simplest assumption
 - The information that will not be needed for the longest time.
 - Is the information that has not been accessed for the longest time.

The “Clock Algorithm”

- LRU is (perceived to be) expensive
 - Maintain timestamp for each block.
 - Update and resort blocks on access.
- The “Clock Algorithm” is a less expensive approximation.
 - Arrange the frames (places blocks can go) into a logical circle like
 - Each frame is marked 0 or 1.
 - Set to 1 when block added to frame.
 - Or when application accesses a block in frame.
 - Replacement choice
 - Sweep second hand clockwise one frame at a time.
 - If bit is 0, choose for replacement.
 - If bit is 1, set bit to zero and go to next frame.
- The basic idea is. On a clock face
 - If the second hand is currently at 27 seconds.
 - The 28 second tick mark is “the least recently touched mark.”



Replacement Algorithm

The algorithms are more sophisticated in the real world, e.g.

- “Scans” are common, e.g. go through a large query result in order (will be more clear when discussing cursors).
 - The engine knows the current position in the result set.
 - Uses the sort order to determine which records will be accessed soon.
 - Tags those blocks as not replaceable.
 - (A form of clairvoyance).
- Not all users/applications are equally “important.”
 - Classify users/applications into priority 1, 2 and 3.
 - Sub-allocate the buffer pool into pools P1, P2 and P3.
 - Apply LRU within pools and adjust pool sizes based on relative importance.
 - This prevents
 - A high access rate, low-priority application from taking up a lot of frames
 - Result in low access, high priority applications not getting buffer hits.



Optimization of Disk Block Access (Cont.)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in Chapter 19)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM or flash buffer immediately
 - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
 - Used exactly like nonvolatile RAM
 - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NV-RAM or log disk
 - Reordering without journaling: risk of corruption of file system data

Indexes



Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in sorted order
 - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.



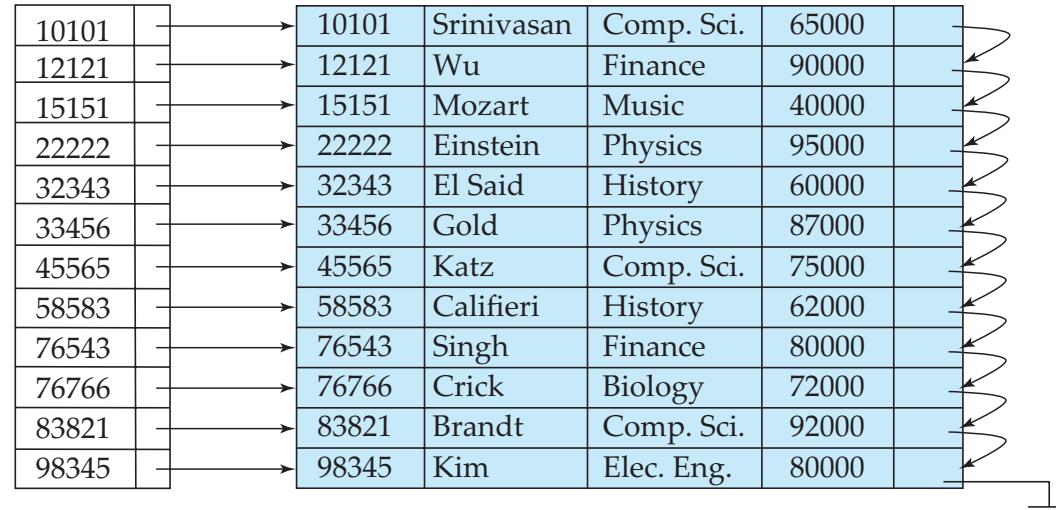
Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **primary index**
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index**.
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.



Dense Index Files

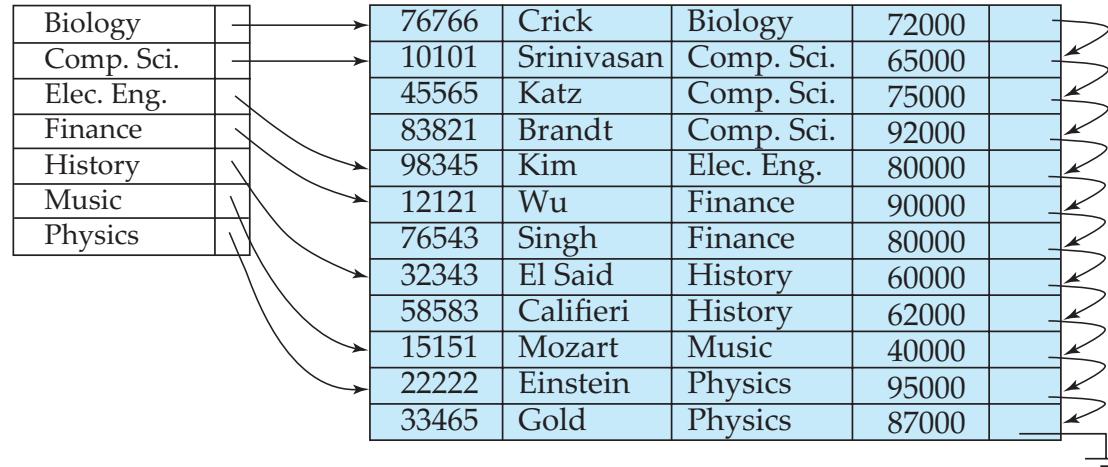
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





Dense Index Files (Cont.)

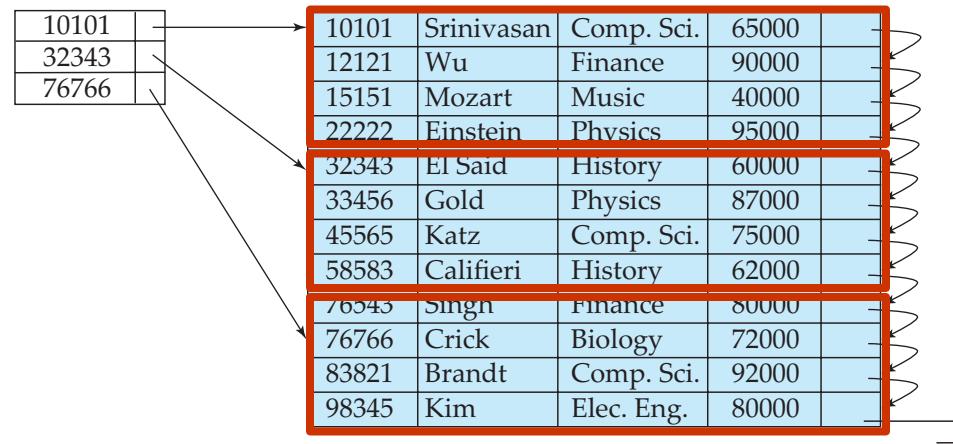
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

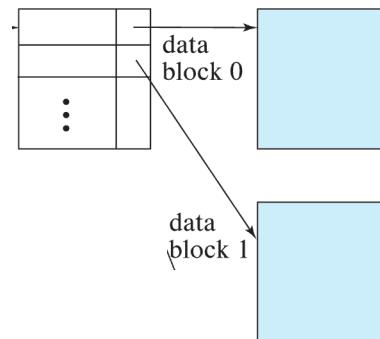
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points





Sparse Index Files (Cont.)

- Compared to dense indices:
 - Less space and less maintenance overhead for insertions and deletions.
 - Generally slower than dense index for locating records.
- **Good tradeoff:**
 - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

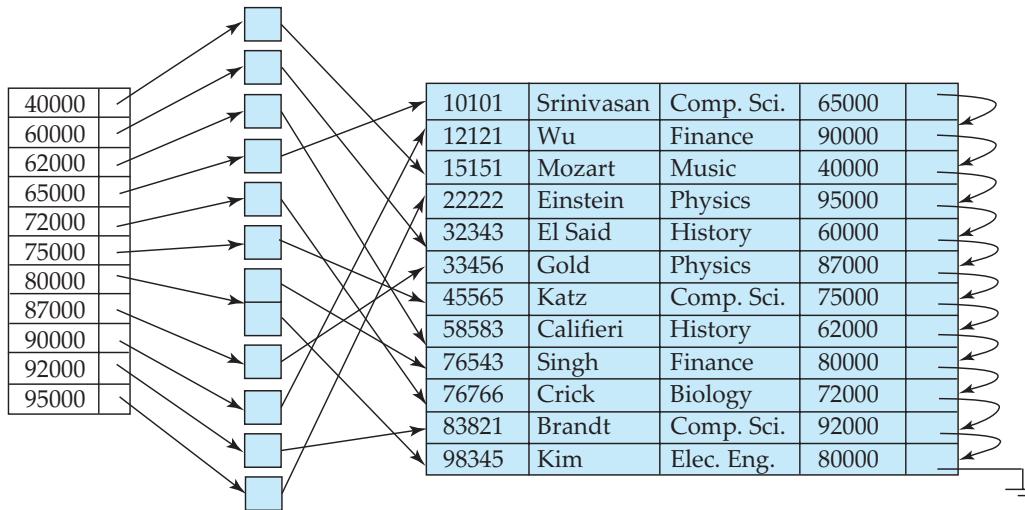


- For unclustered index: sparse index on top of dense index (multilevel index)



Secondary Indices Example

- Secondary index on salary field of instructor



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

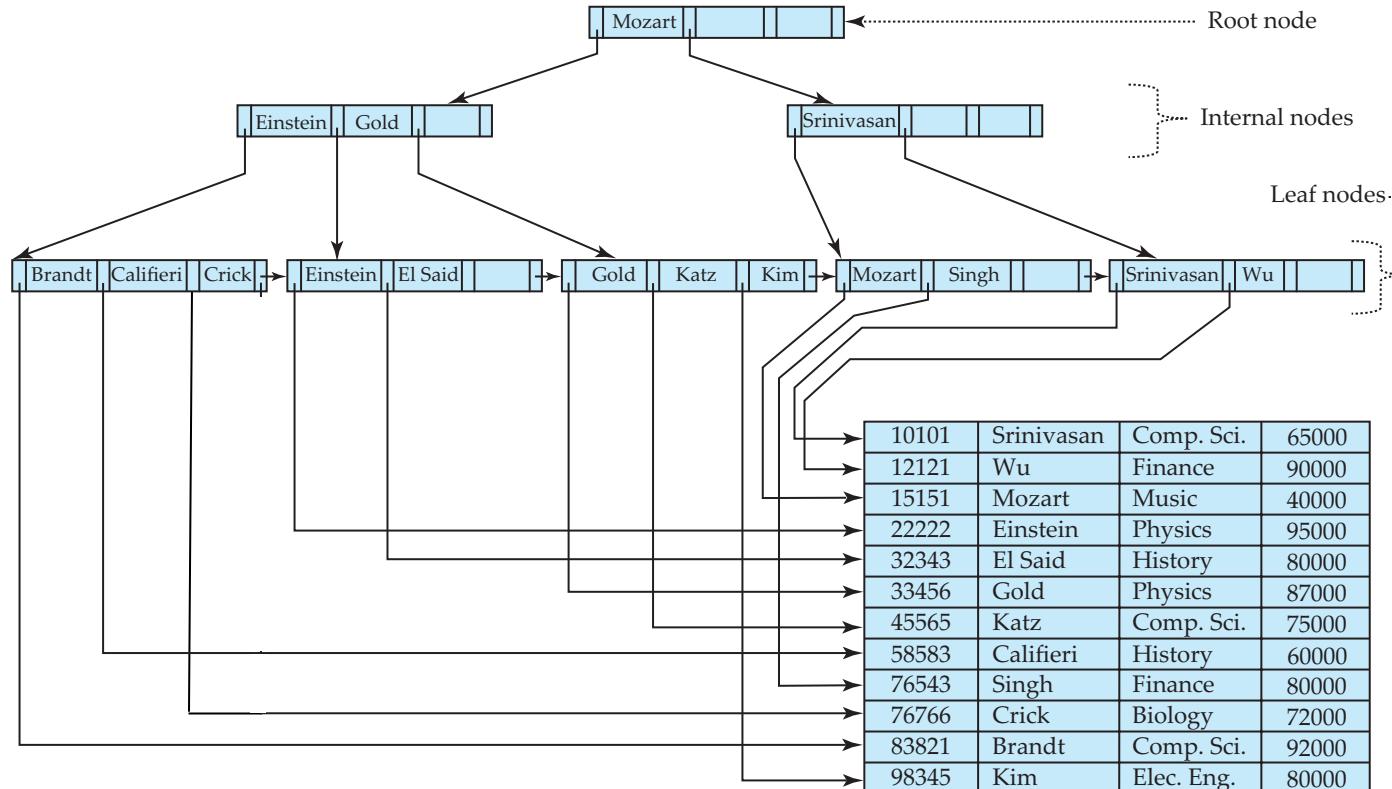


Indices on Multiple Keys

- **Composite search key**
 - E.g., index on *instructor* relation on attributes (*name*, *ID*)
 - Values are sorted lexicographically
 - E.g. (John, 12121) < (John, 13514) and (John, 13514) < (Peter, 11223)
 - Can query on just *name*, or on (*name*, *ID*)
- (nameLast, nameFirst, birthyear)
 - nameLast [nameLast = “Ferguson”] [nameLast like “Fer%”]
 - nameLast, nameFirst
 - nameLast, nameFirst, birthyear
- NOT and index on
 - nameFirst, nameLast
 - birthyear
 - nameLast like [%er%]



Example of B⁺-Tree





B⁺-Tree Index Files (Cont.)

A B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - If the root is not a leaf, it has at least 2 children.
 - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.



B⁺-Tree Node Structure

- Typical node

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

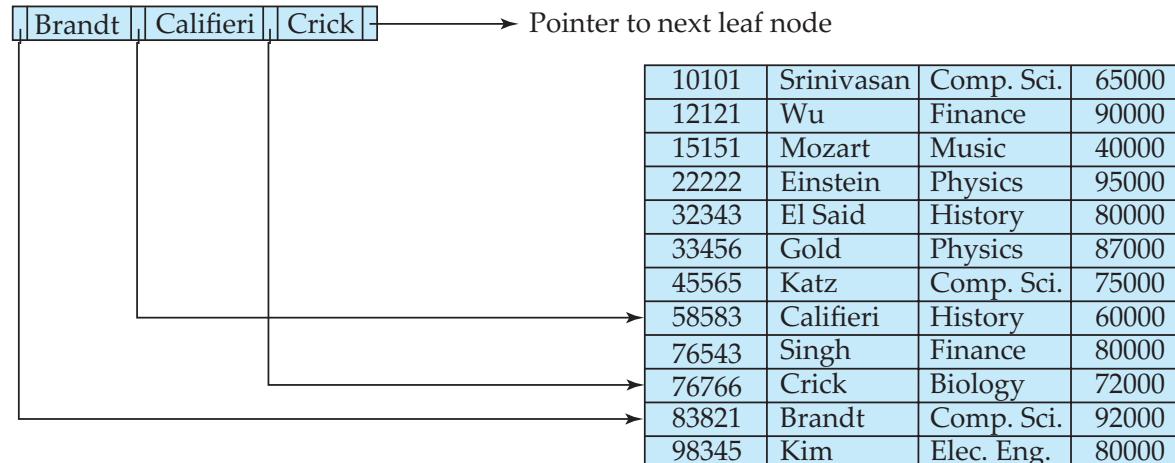
(Initially assume no duplicate keys, address duplicates later)



Leaf Nodes in B⁺-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i points to a file record with search-key value K_i ,
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than or equal to L_j 's search-key values
- P_n points to next leaf node in search-key order





Non-Leaf Nodes in B⁺-Trees

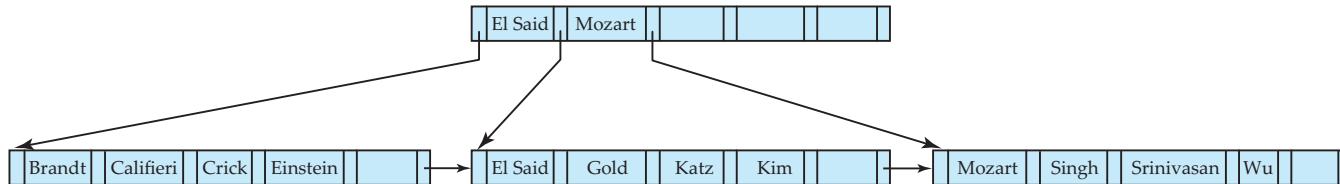
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
 - All the search-keys in the subtree to which P_1 points are less than K_1
 - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
 - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
 - General structure

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------



Example of B⁺-tree

- B⁺-tree for *instructor* file ($n = 6$)



- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2) \rceil$ and n with $n = 6$).
- Root must have at least 2 children.



Observations about B+-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
 - Level below root has at least $2 * \lceil n/2 \rceil$ values
 - Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ values
 - .. etc.
 - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
 - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Show the Simulator

<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>



Hashing



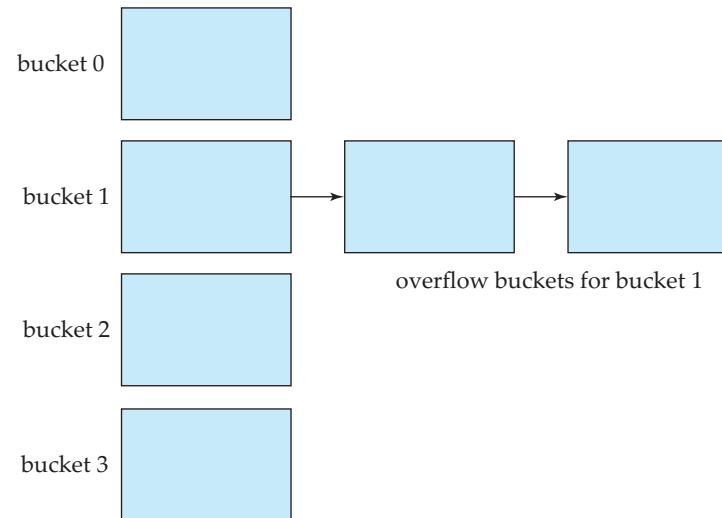
Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
 - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records



Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing** (also called **closed hashing** or **open hashing** depending on the book you use)
 - An alternative, called **open addressing** (also called **open hashing** or **closed hashing** depending on the book you use) which does not use overflow buckets, is not suitable for database applications.





Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7



Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
 - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
 - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
 - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Show the Simulator

<http://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html>
<https://opendsa-server.cs.vt.edu/ODSA/AV/Development/hashAV.html>

NoSQL

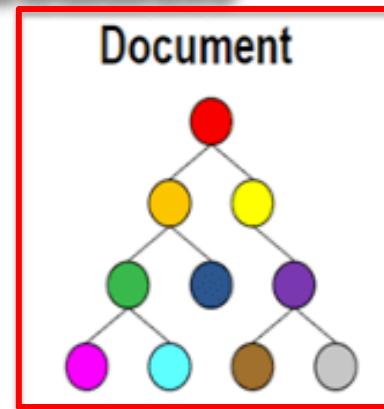
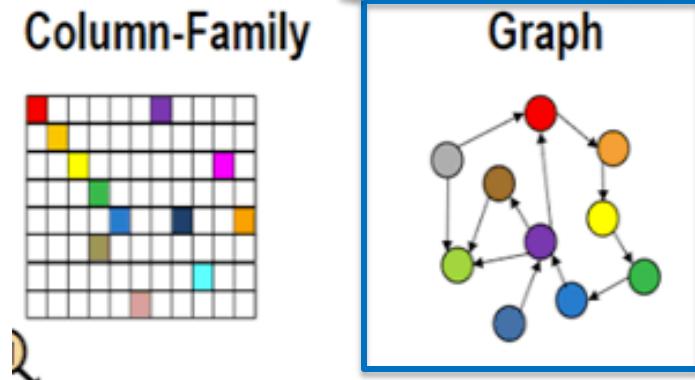
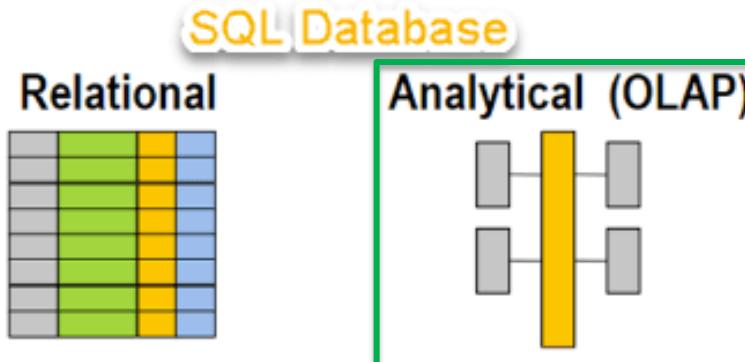
Reminder

Simplistic Classification

(<https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b>)

Relational is the foundational model.

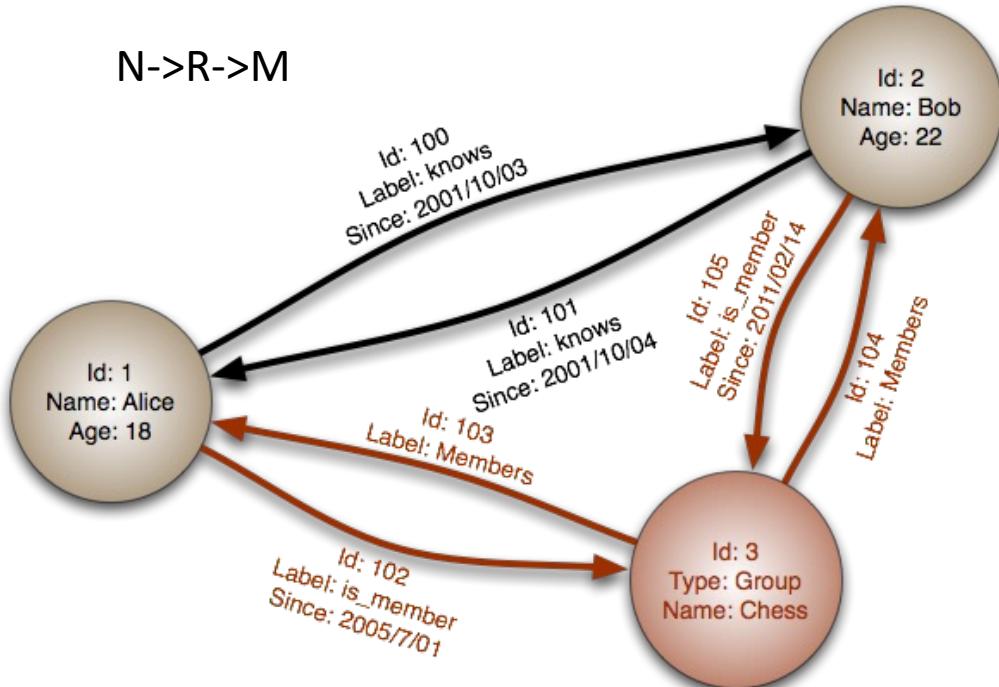
We covered graphs and examples.



Graph Databases

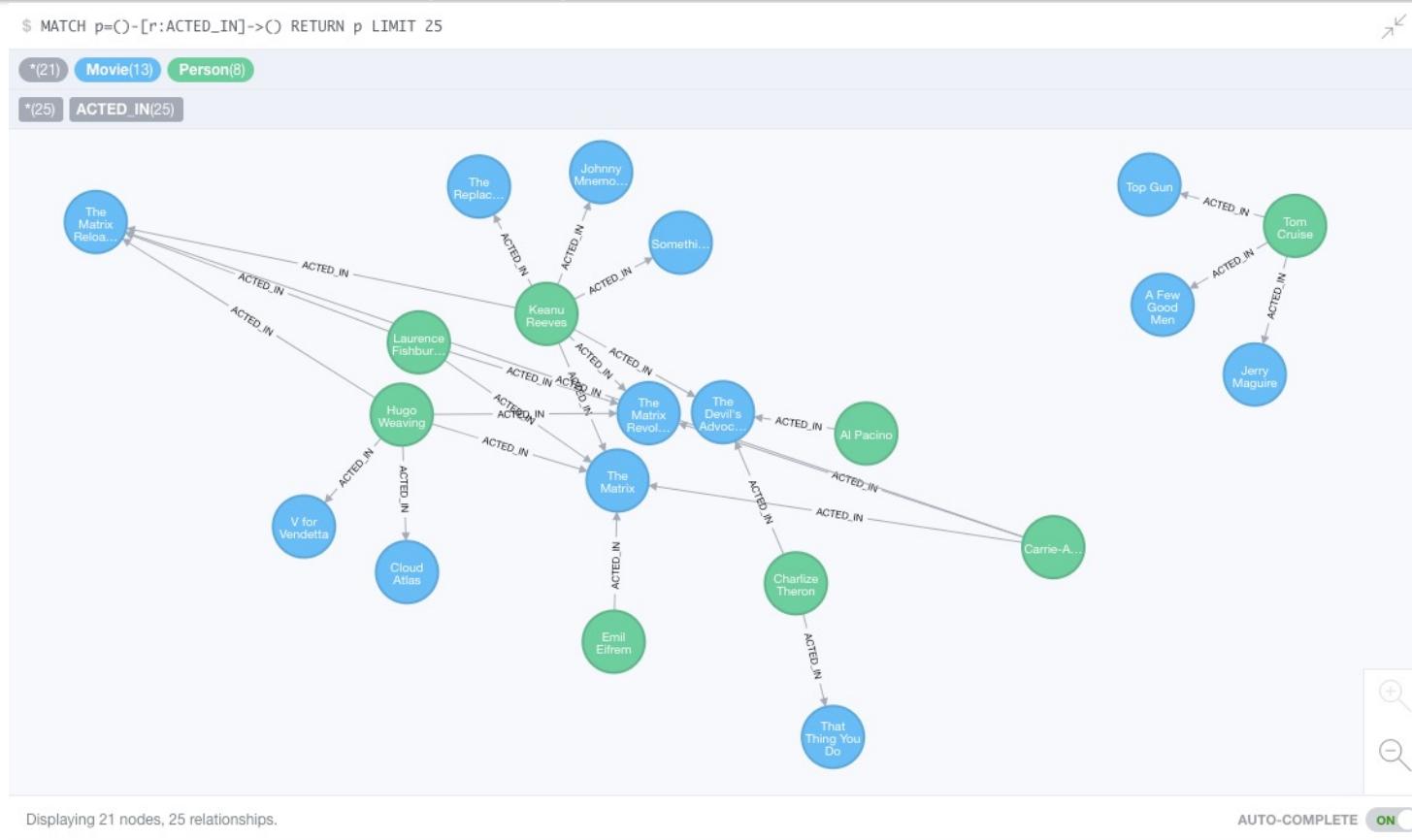
Graph Database

- Exactly what it sounds like
- Two core types
 - Node
 - Edge (link)
- Nodes and Edges have
 - Label(s) = “Kind”
 - Properties (free form)
- Query is of the form
 - $p1(n)-p2(e)-p3(m)$
 - n, m are nodes; e is an edge
 - $p1, p2, p3$ are predicates on labels



Neo4J Graph Query

```
$ MATCH p=(:Movie)-[r:ACTED_IN]->(:Person) RETURN p LIMIT 25
```



Why Graph Databases?

- Schema Less and Efficient storage of Semi Structured Information
- No O/R mismatch – very natural to map a graph to an Object Oriented language like Ruby.
- Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
- Very natural to express graph related problem with traversals (recommendation engine, find shortest path etc..)
- Seamless integration with various existing programming languages.
- ACID Transaction with rollbacks support.
- Whiteboard friendly – you use the language of node, properties and relationship to describe your domain (instead of e.g. UML) and there is no need to have a complicated O/R mapping tool to implement it in your database. You can say that Neo4j is “Whiteboard friendly” !(<http://video.neo4j.org/JHU6F/live-graph-session-how-allison-knows-james/>)

Social Network “path exists” Performance

- Experiment:
 - ~1k persons
 - Average 50 friends per person
 - `pathExists(a, b)` limited to depth 4

	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

Graph databases are

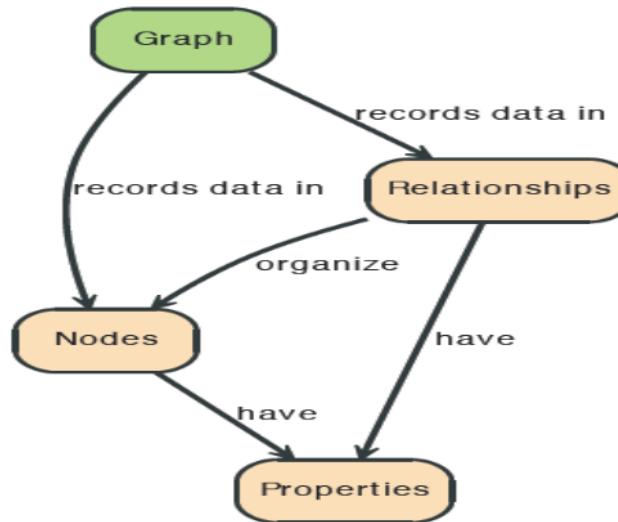
- Extremely fast for some queries and data models.
- Implement a language that vastly simplifies writing queries.

What are graphs good for?

- Recommendations
- Business intelligence
- Social computing
- Geospatial
- Systems management
- Web of things
- Genealogy
- Time series data
- Product catalogue
- Web analytics
- Scientific computing (especially bioinformatics)
- Indexing your *slow* RDBMS
- And much more!

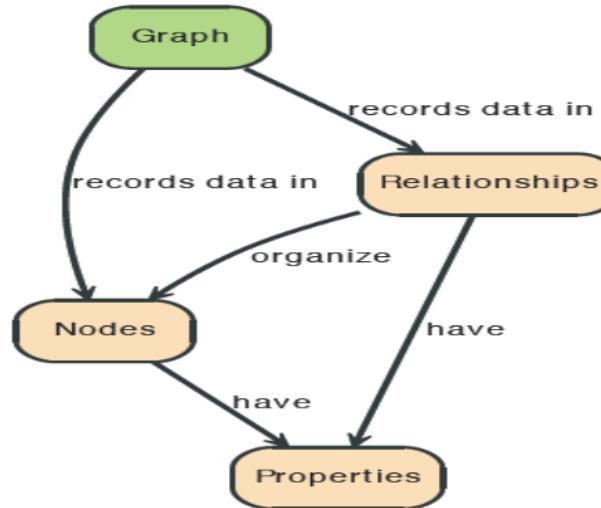
Graphs

- “A Graph —records data in → Nodes —which have → Properties”



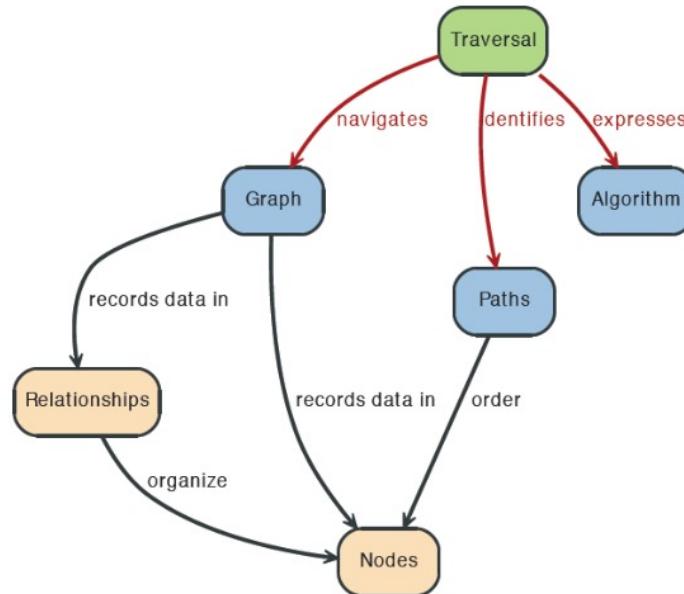
Graphs

- “Nodes —are organized by → Relationships — which also have → Properties”



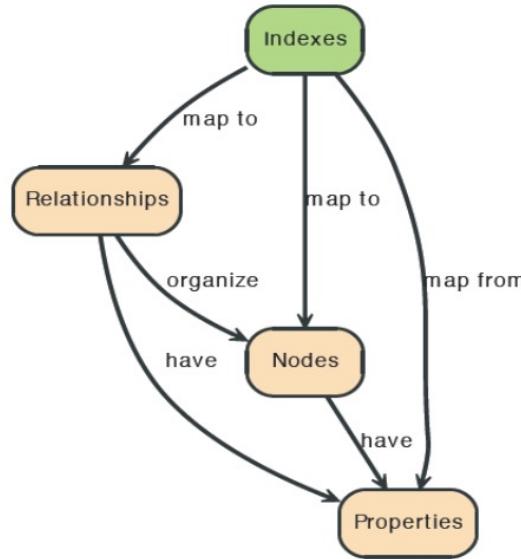
Query a graph with Traversal

- “A Traversal —navigates→ a Graph; it — identifies→ Paths —which order→ Nodes”



Indexes

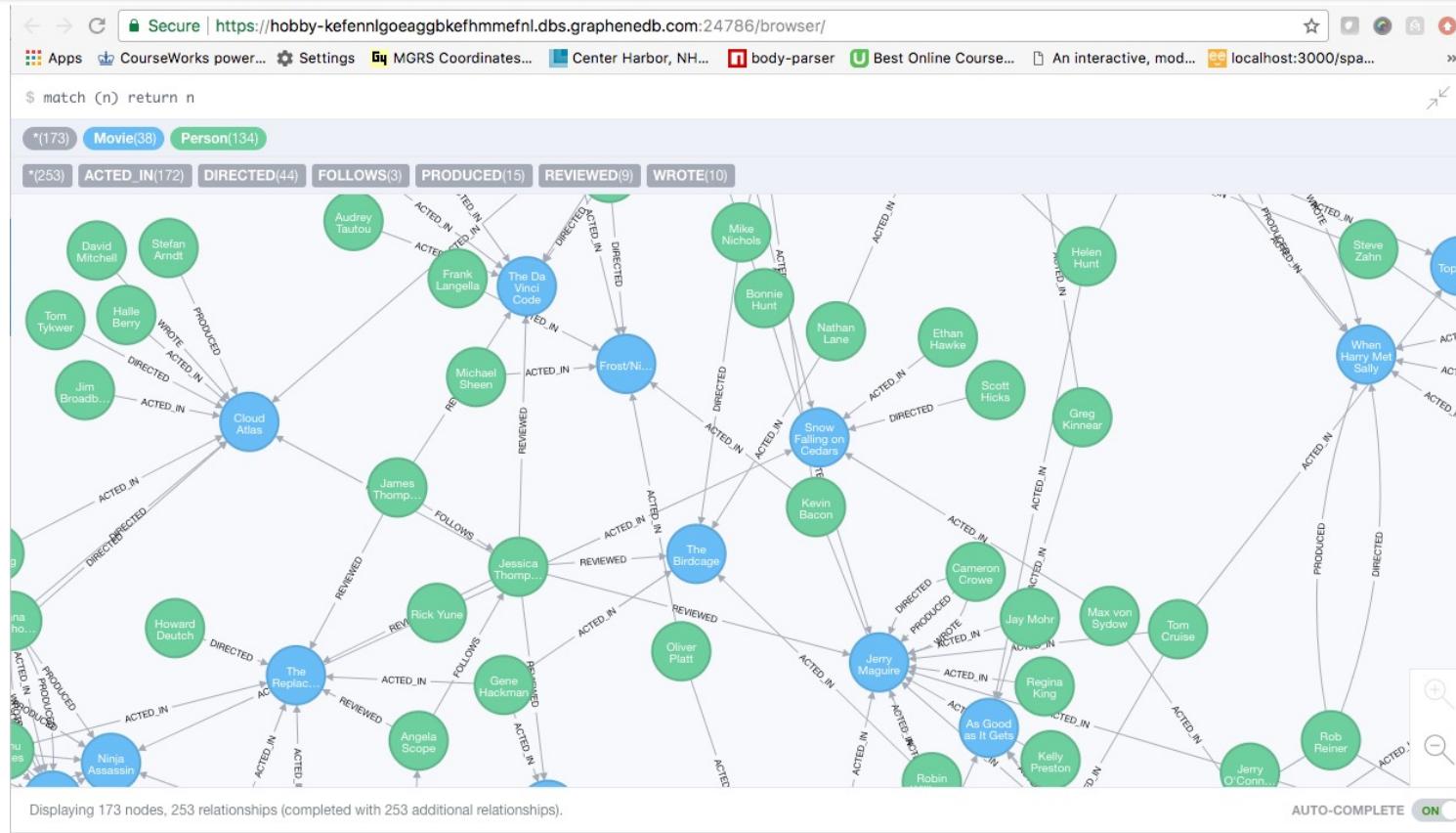
- “An Index —maps from → Properties —to either → Nodes or Relationships”



Demo

- Demo online DB

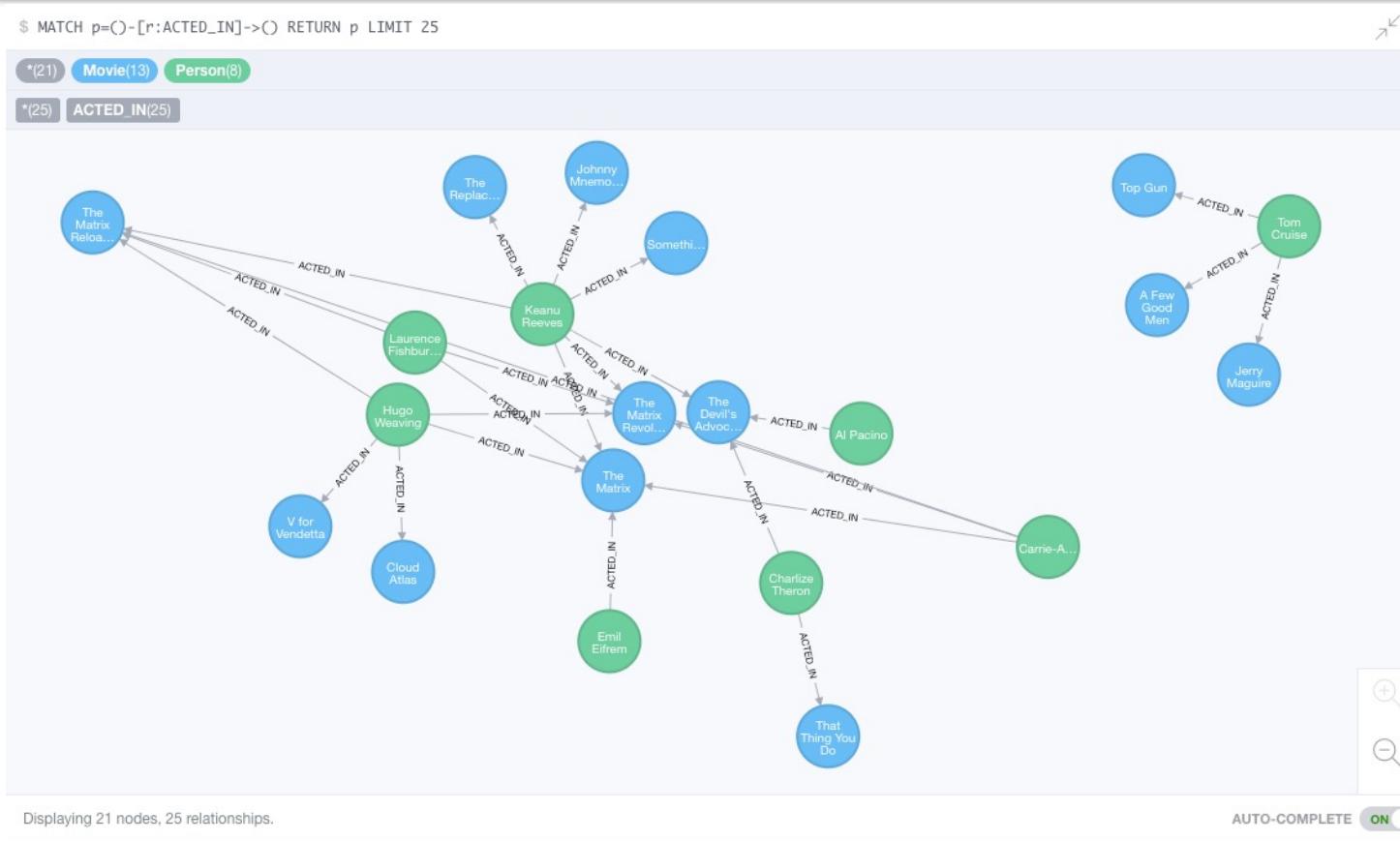
A Graph Database (Sample)



Neo4J Graph Query

Who acted in which movies?

```
$ MATCH p=(n)-[r:ACTED_IN]->(m) RETURN p LIMIT 25
```



Big Deal. That is just a JOIN.

- Yup. But that is simple.
- Try writing the queries below in SQL.

The Movie Graph Recommend

Let's recommend new co-actors for Tom Hanks. A basic recommendation approach is to find connections past an immediate neighborhood which are themselves well connected.

For Tom Hanks, that means:

1. Find actors that Tom Hanks hasn't yet worked with, but his co-actors have.
2. Find someone who can introduce Tom to his potential co-actor.

Extend Tom Hanks co-actors, to find co-co-actors who haven't work with Tom Hanks...

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]-(m)<-[ACTED_IN]-(coActors),  
      (coActors)-[:ACTED_IN]-(m2)<-[ACTED_IN]-(cocoActors)  
WHERE NOT (tom)-[:ACTED_IN]-(m2)  
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

Find someone to introduce Tom Hanks to Tom Cruise

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]-(m)<-[ACTED_IN]-(coActors),  
      (coActors)-[:ACTED_IN]-(m2)<-[ACTED_IN]-(cruise:Person {name:"Tom Cruise"})  
RETURN tom, m, coActors, m2, cruise
```

Recommend

```
1 MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),  
2     (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)  
3 WHERE NOT (tom)-[:ACTED_IN]->(m2)  
4 RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```



```
$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors), (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors) ...
```



	Recommended	Strength
Rows	Tom Cruise	5
A	Zach Grenier	5
Text	Helen Hunt	4
</>	Cuba Gooding Jr.	4
Code	Keanu Reeves	4
	Tom Skerritt	3
	Carrie-Anne Moss	3
	Val Kilmer	3
	Bruno Kirby	3
	Philip Seymour Hoffman	3
	Billy Crystal	3
	Carrie Fisher	3

```

1 MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
2   (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
3 RETURN tom, m, coActors, m2, cruise

```



\$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors), (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})



Graph

*(13) Movie(8) Person(5)

Rows

A Text

</> Code



Which actors have worked with both Tom Hanks and Tom Cruise?

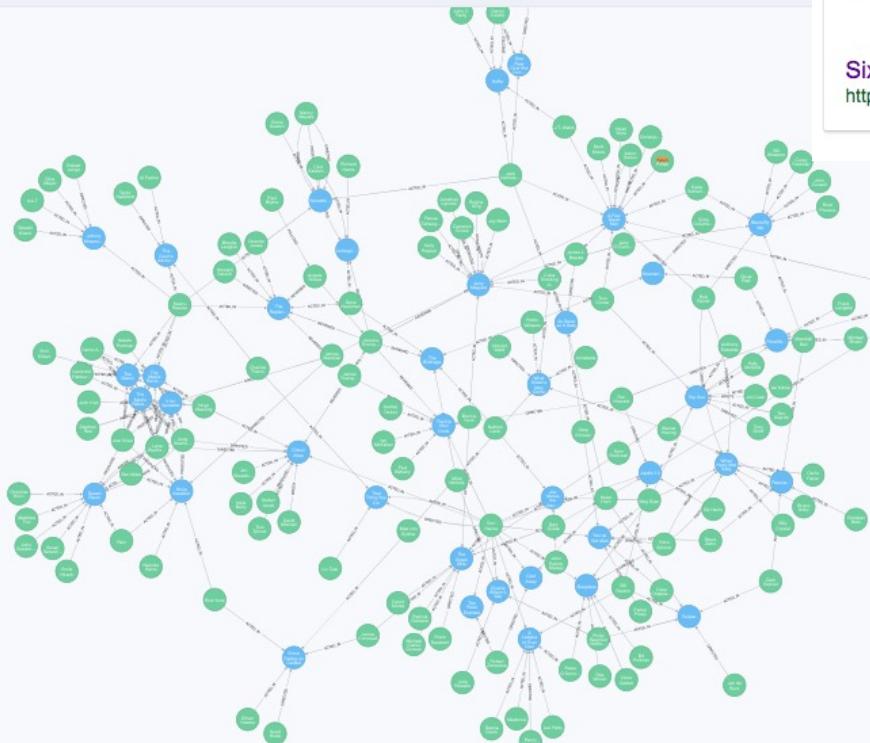
Displaying 13 nodes, 16 relationships (completed with 16 additional relationships).

AUTO-COMPLETE

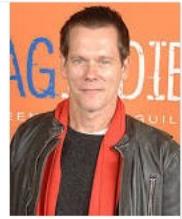
```
$ MATCH (s:Person { name: 'Kevin Bacon' })-[*0..6]-(m) return s,m
```

*(171) Movie(38) Person(133)

(253) ACTED_IN(172) DIRECTED(44) FOLLOWS(3) PRODUCED(15) REVIEWED(9) WROTE(10)



Six Degrees of Kevin Bacon is a parlour game based on the "six degrees of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart. Movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific actor **Kevin Bacon**.



Six Degrees of Kevin Bacon - Wikipedia
https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

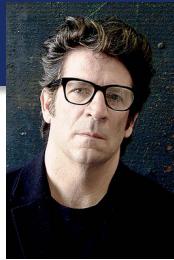
About this result

Feedback

Six Degrees of Kevin Bacon

Game





How do you get from Kevin Bacon to Robert Longo?

```
$ MATCH (kevin:Person { name: 'Kevin Bacon' }), (robert:Person { name: 'Robert Longo' }), p = shortestPath((kevin)-[*..15]-(robert)) RETURN p
```

