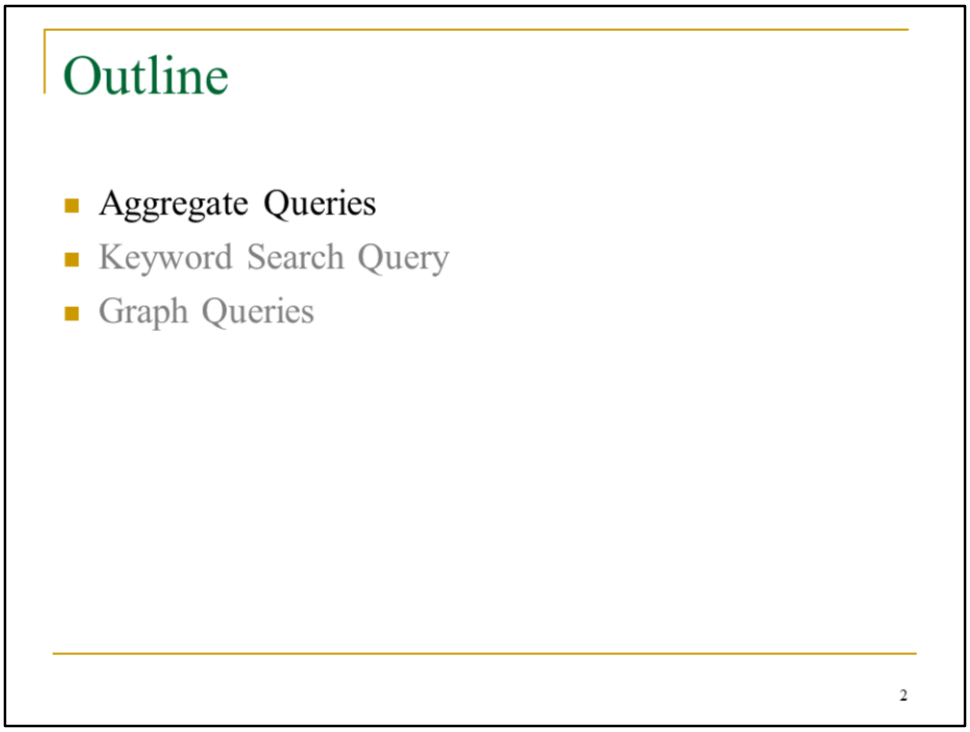
Big Data Analytics

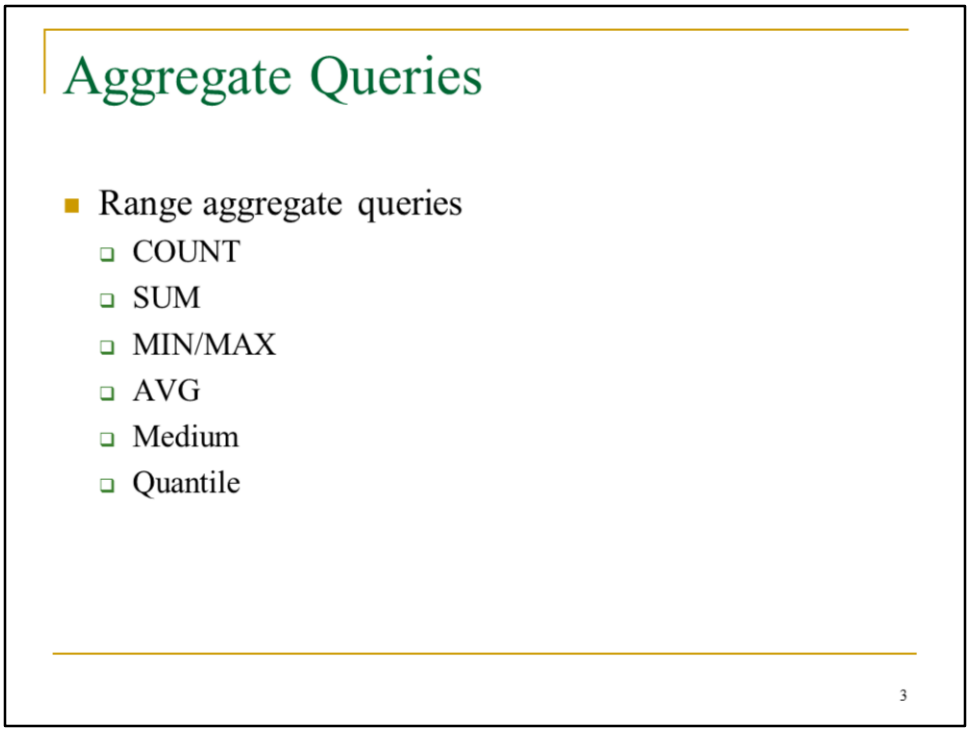
Chapter 9: Queries Over Big Data (Part 3)

1

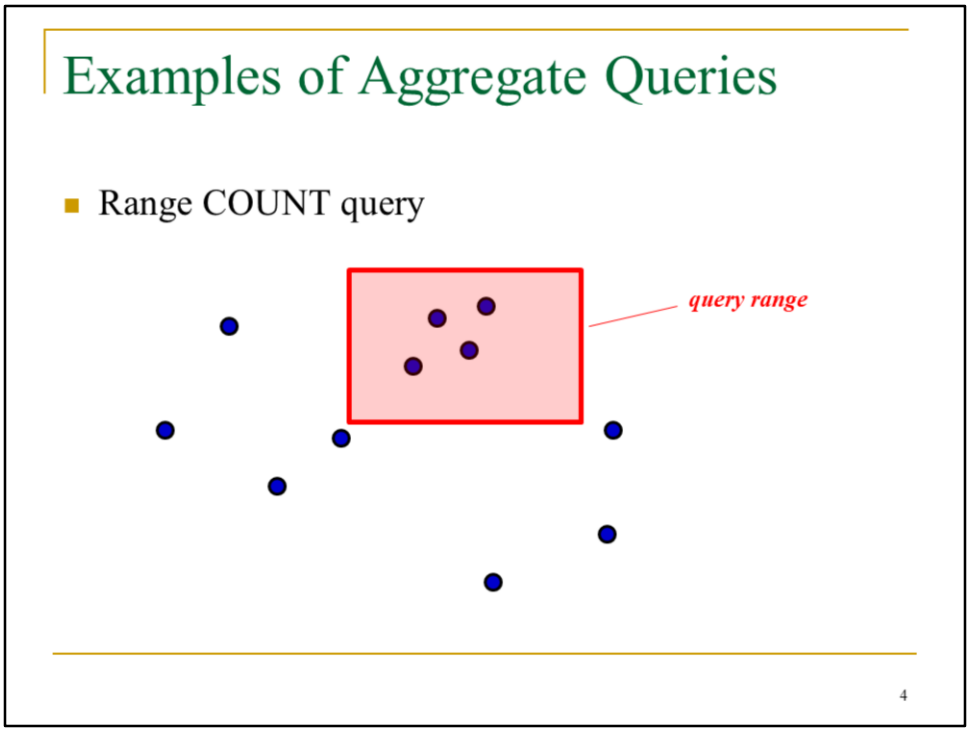
Chapter 9: Queries Over Big Data (Part 3)
In this chapter, we will continue to discuss more queries over big data.



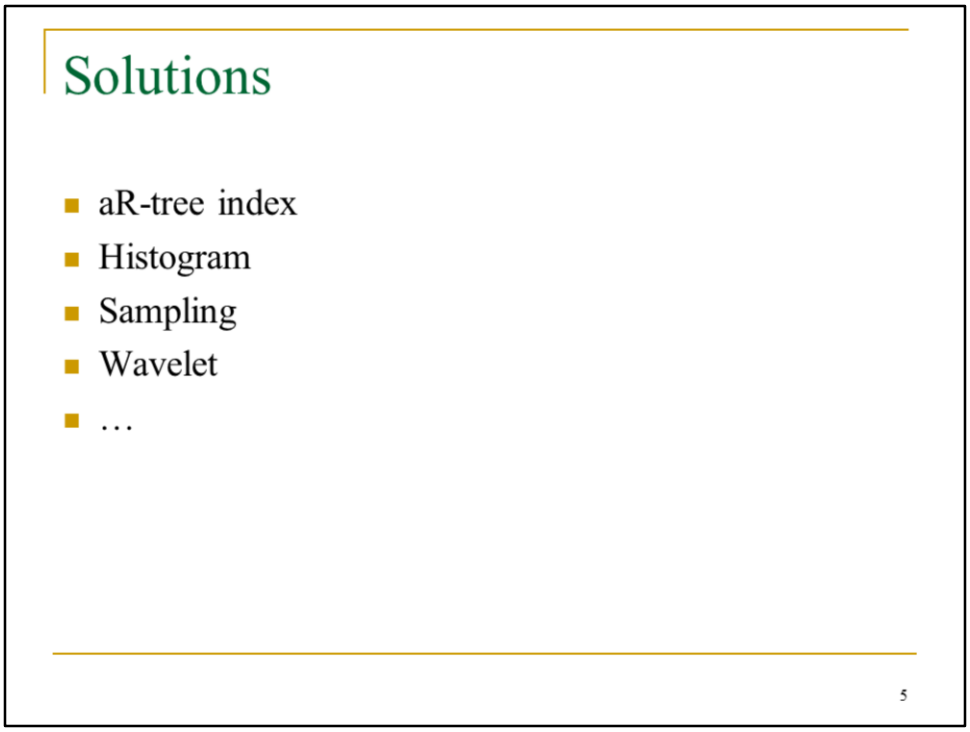
First, we will talk about aggregate queries.



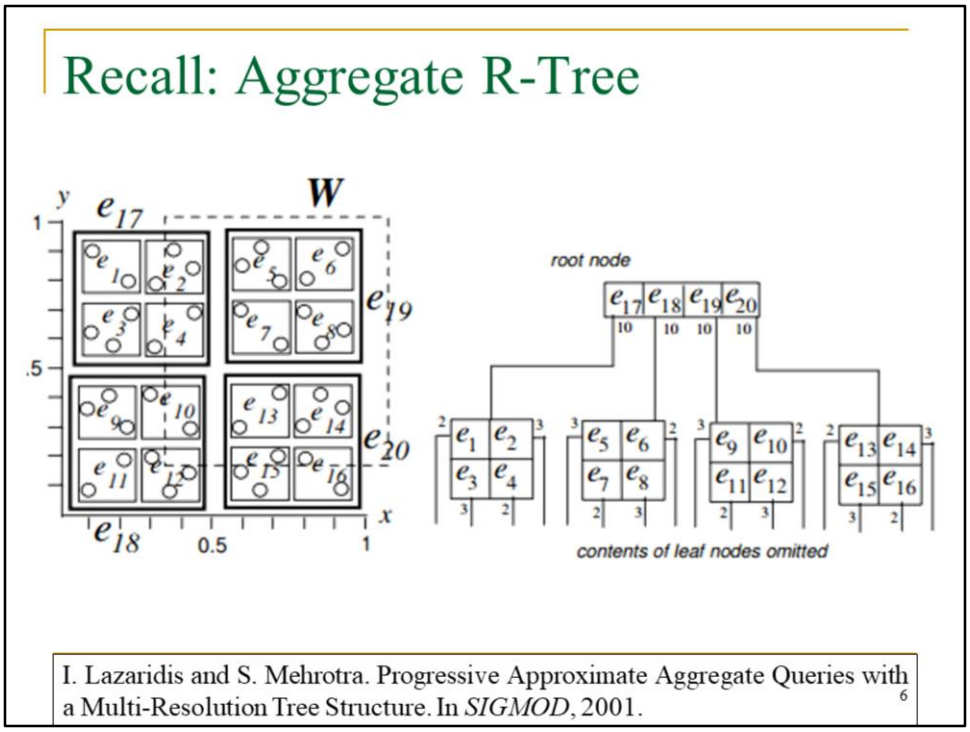
There are many aggregate query types, such as COUNT, SUM, MIN/MAX, AVG, medium, and quantile.



As an example, for range COUNT query, given a query range, we want to retrieve the number of objects falling into this range.

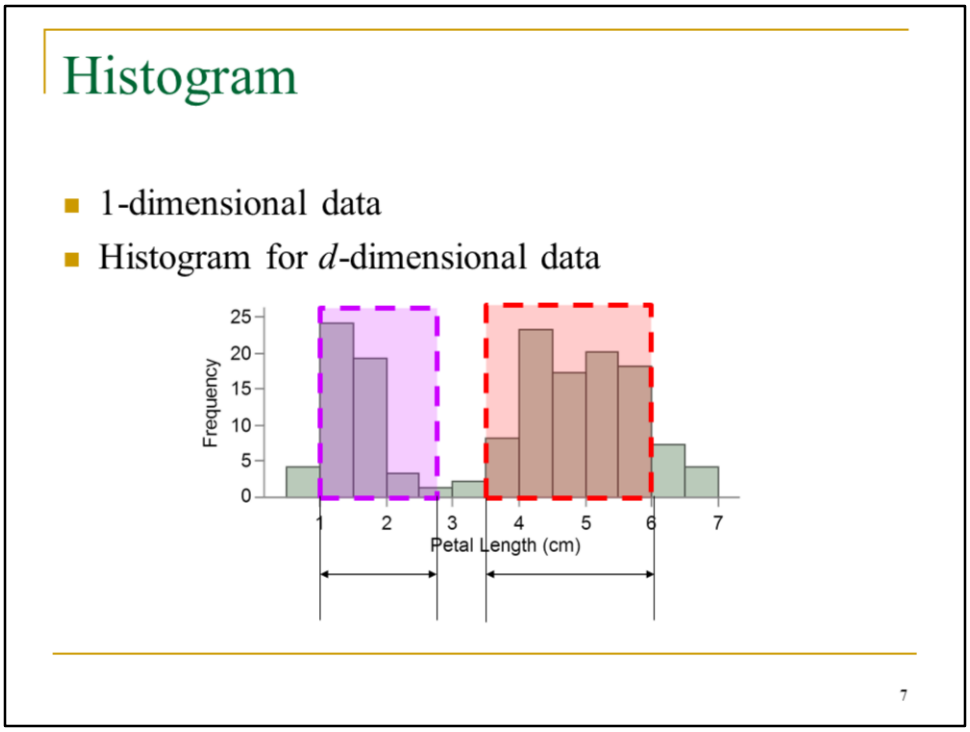


There are different possible solutions to aggregate problems, exact and approximate solutions. For the exact solution, we will discuss how to use aR-tree index to retrieve aggregate answers. For approximate solutions, there are some estimation methods such as histogram, sampling, and wavelet.



For the exact solution, we can traverse the aR-tree to obtain aggregate answers. Recall that we talked about the aR-tree index, in which each intermediate node stores some aggregate information (e.g., COUNT, SUM, etc.).

For range COUNT query, we can start from the root of the aR-tree, and visit those nodes whose MBRs intersect with the query range. If we encounter an MBR node completely inside the query range, then we can directly use its associated COUNT value, without accessing its underlying objects; if we encounter an MBR node partially intersecting with the query range, then we need to further check its children; if an MBR node does not intersect with the query range, then we can simply ignore this node. After the index traversal, we can obtain the exact number of objects inside the query range.



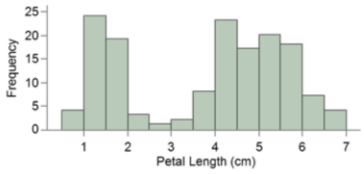
Another direction of the aggregate query is to use synopses to estimate the approximate aggregate values within an error bound. One type of synopsis is the histogram. In the example of the figure, we show a histogram built over 1-dimensional data. We divide the value domain [0, 7] into 14 buckets, and each bucket is associated with the frequency of data falling into this bucket.

For a given query range [3.5, 6], since this query range exactly covers 5 buckets, we can retrieve the exact COUNT value by summing up frequencies of these 5 buckets.

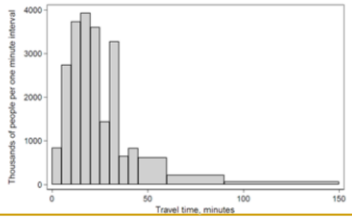
For another query range [1, 2.75], it only covers half of a bucket [2.5, 3]. In this case, we can only assume data are uniformly distributed inside the bucket, and estimate the frequency of the range [2.5, 2.75], by taking half of frequency in this bucket.

Categories of Histograms

- Equi-width histogram

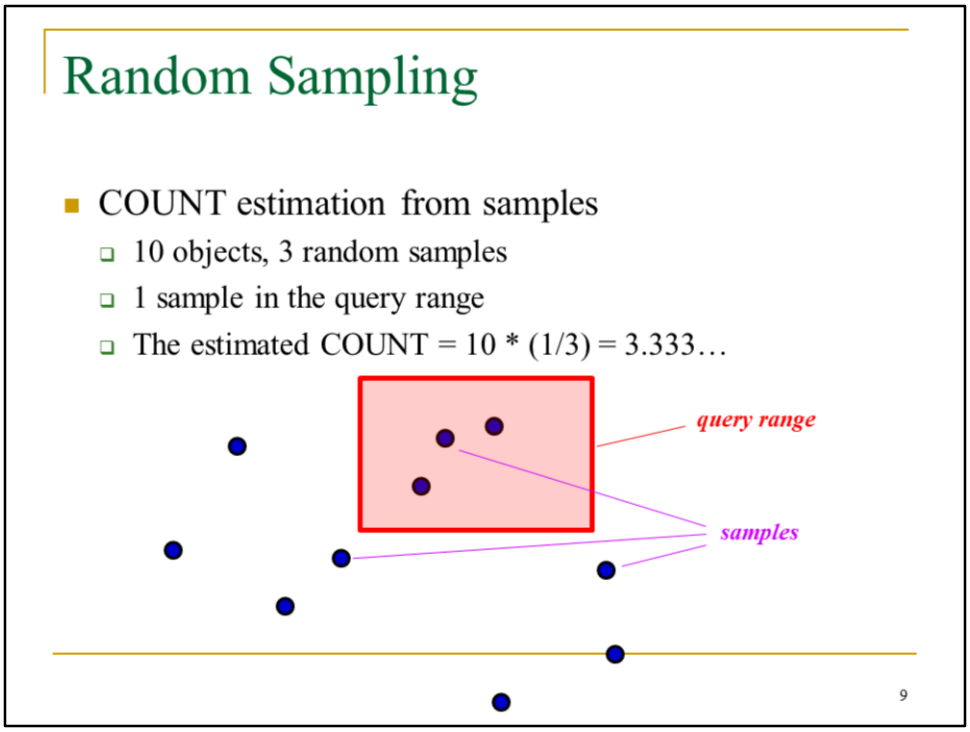


- Equi-depth histogram

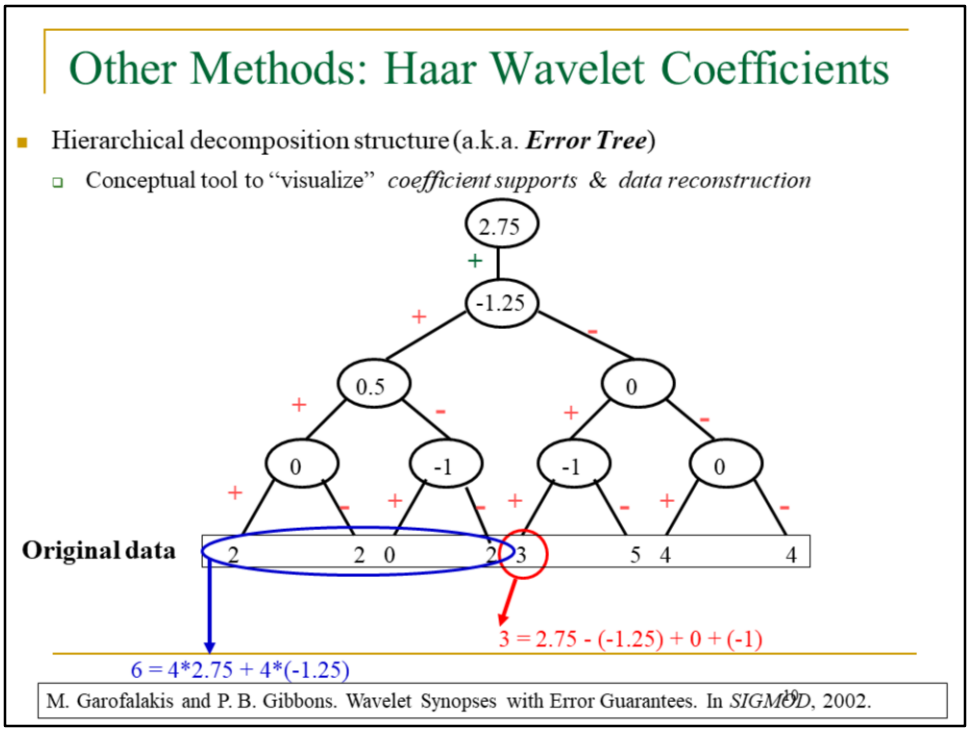


8

In the literature, there two types of histograms, equi-width and equi-depth. For the equi-width histogram, we divide the data space into buckets of the same width. For the equi-depth histogram, we divide the data space into buckets of different sizes such that all buckets store approximately the same number of objects.



Another direction is to consider sampling method. As shown in the figure, there are 10 objects (blue points) in the data space. We can randomly obtain 3 samples out of 10 objects. Given a query range, only one out of 3 samples is falling into the query range. So we can estimate the number of objects inside the query range by using 10 * 1/3 = 3.3333, which is close to the exact COUNT aggregate 3. Intuitively, due to the random sampling, the chance that each object falls into the query range is given by 1/3. Thus, 10 * 1/3 is the expected number of objects inside the query range.



There is another synopsis for summarizing numeric data, called Haar wavelet. As shown in the figure, we have 8 original data items at the bottom of the wavelet. For any node in the tree structure, its value is given by the difference between the average of all values under the left branch and the average of both branches. For example, the third and fourth original data items, 0 and 2, have the average value 1, and their parent is given by 0-1, that is, -1. This way, we can build a hierarchical tree structure. The root of the tree is the average of all the 8 items in the original data sequence.

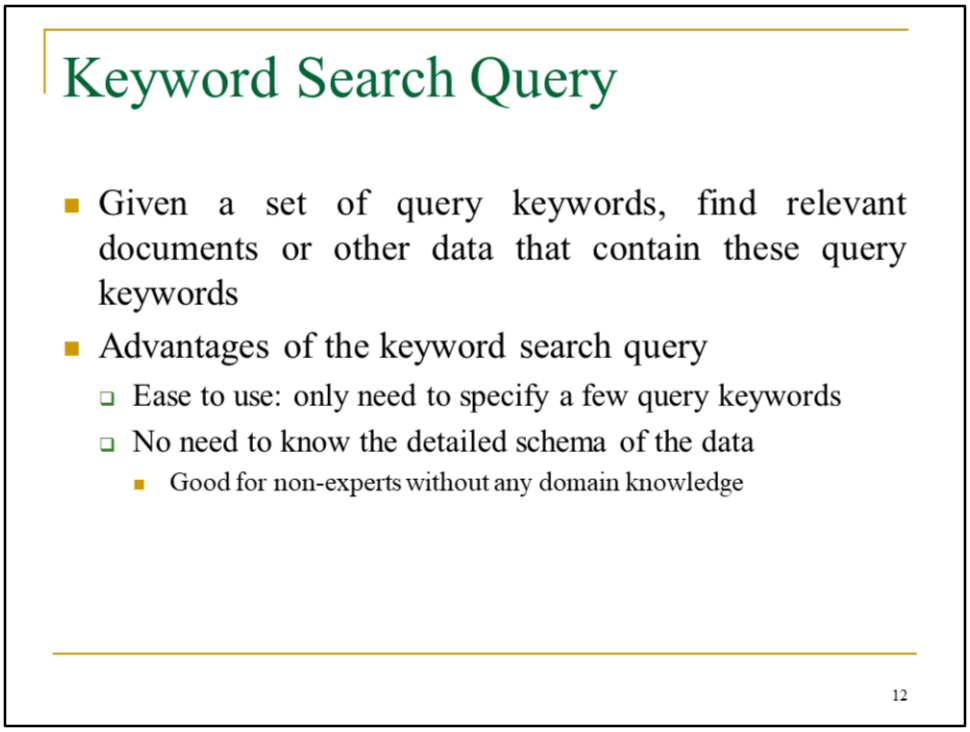
The Haar wavelet coefficients can be used for answering aggregate queries such as AVG or SUM aggregate.

Outline

- Aggregate Queries
- Keyword Search Query
- Graph Queries

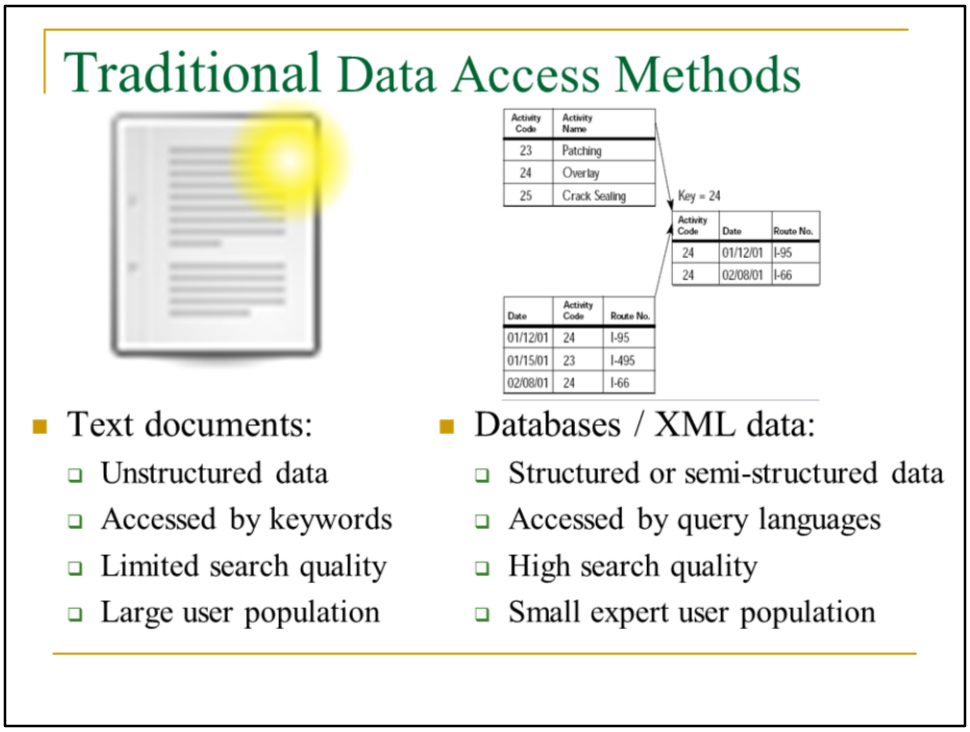
11

Next, we will discuss the keyword search query.



The keyword search query is to return a number of relevant documents and their relationships, such that these documents contain a given set of query keywords. The keyword search is easy to use, that is, users only need specify a few query keywords and the results will be returned. Moreover, those users who are not experts about the underlying data schema can still use the keyword search query. He/she does not need to have the domain knowledge about the structure or meanings of the data.

This type of the query is very similar to a search engine such as Google. But different from the search engine that returns separate URLs/documents, the keyword search query will return a few documents that are highly correlated.



References

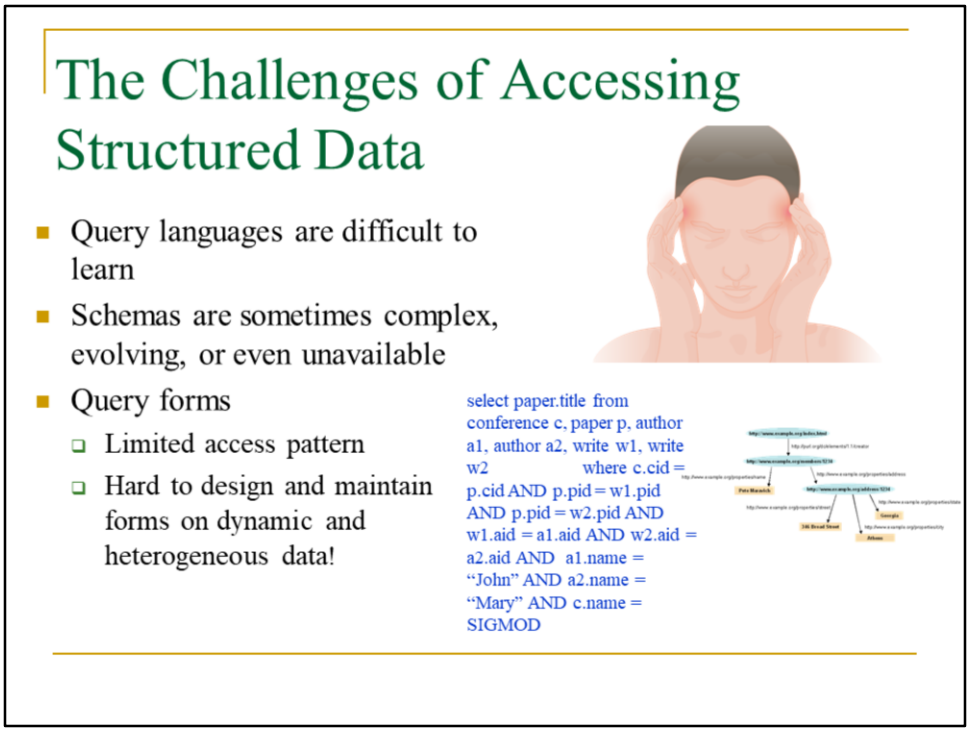
bdesham, The people from the Tango! project, modification by. *An Icon Representing the Creation of a New Text Document.* 4 Sept. 2007. Ripped directly from the Tango Project's File:Document-new.svg and File:Text-x-generic.svg., *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:New_text_document.svg. Changes were not made.

Transportation, U. S. Department of. *Relational Model*. Aug. 2001. Data Integration Glossary., *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:Relational_Model.jpg. Changes were not made.

Traditionally, we usually have many text documents, in which texts are unstructured, and accessed by keywords. The text documents are used by many people. However, it is possible to have many documents containing the same keywords, and the search quality is limited.

Another classic way to manage data is to use the database or XML documents, which models structured or semi-structured data, respectively. Such data storages are usually accessed by query languages such as SQL for relational databases or

XQuery/XPath for XML documents. The search quality is high, however, only users who know SQL, Xquery, or Xpath queries can access the data. Thus, the user population is relatively small, compared with users of text documents.



References

Jagadish, H. V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A., and Yu, C. (2007). Making database systems usable. In SIGMOD, pages 13-24.

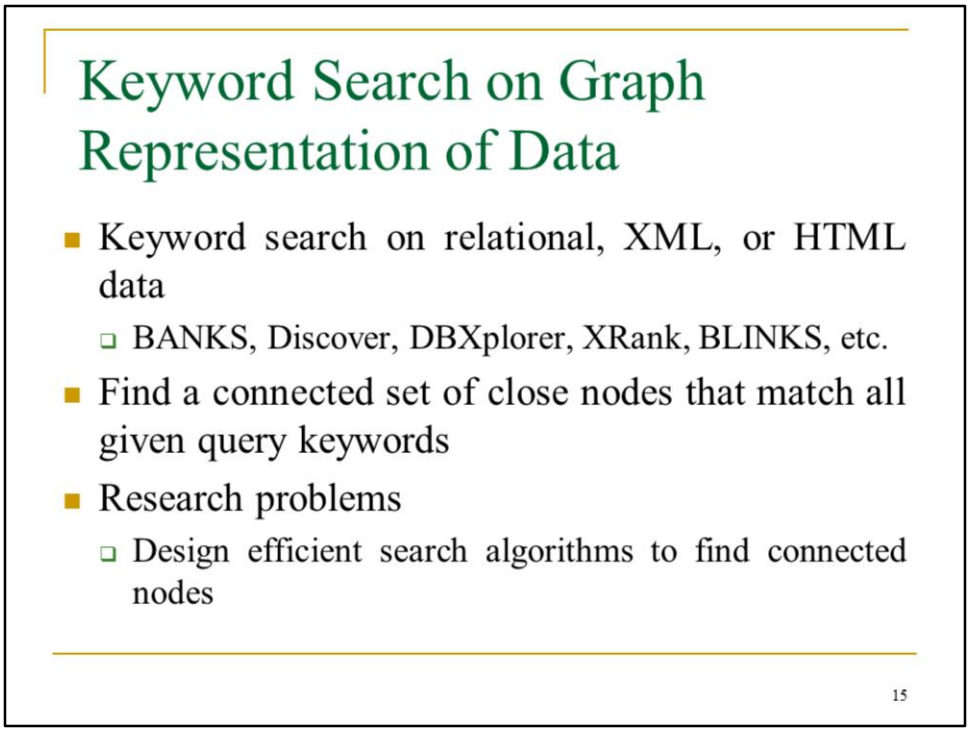
Injurymap. *Dansk: Man Med Hovepine*. 8 Jan. 2019. https://www.injurymap.com/free-human-anatomy-illustrations, *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:Headache.png. Changes were not made.

File:Rdf-Graph2.Png - Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Rdf-graph2.png. Accessed 10 Mar. 2019. Changes were not made.

To handle the structured data, it is challenging for users to learn a new query language, which has a long learning curve. Users need to understand the underlying data schema, which is complex, evolving or sometimes even unavailable.

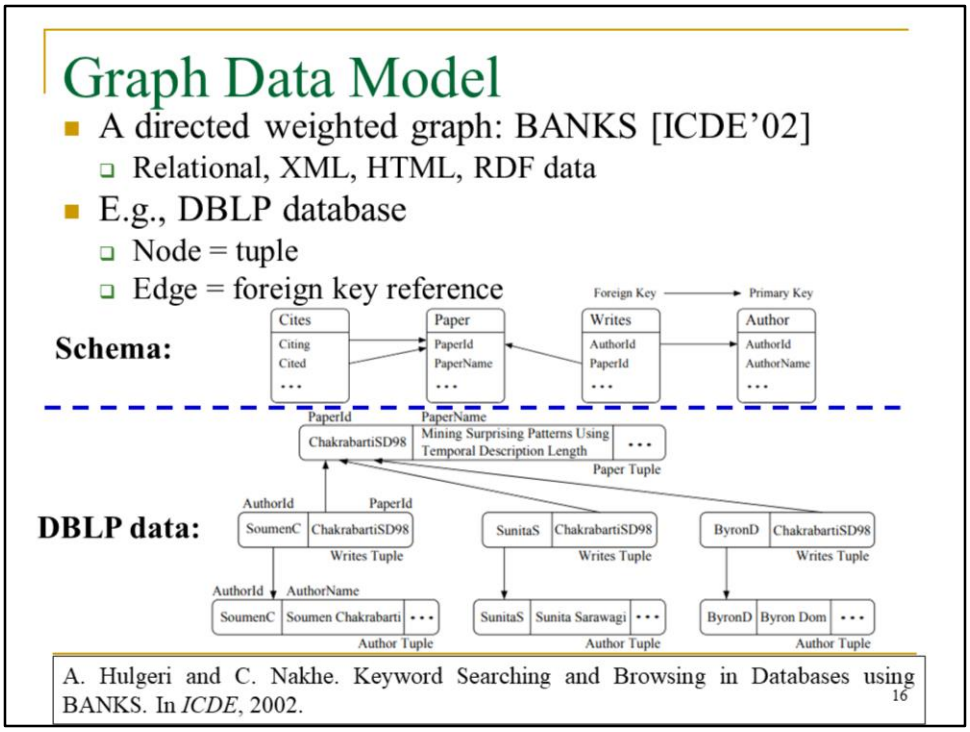
Another possible way is to ask users to fill in some pre-defined query forms, which are however not very flexible. These forms usually have limited access patterns, and

hard to design or maintain for dynamic and heterogeneous data.

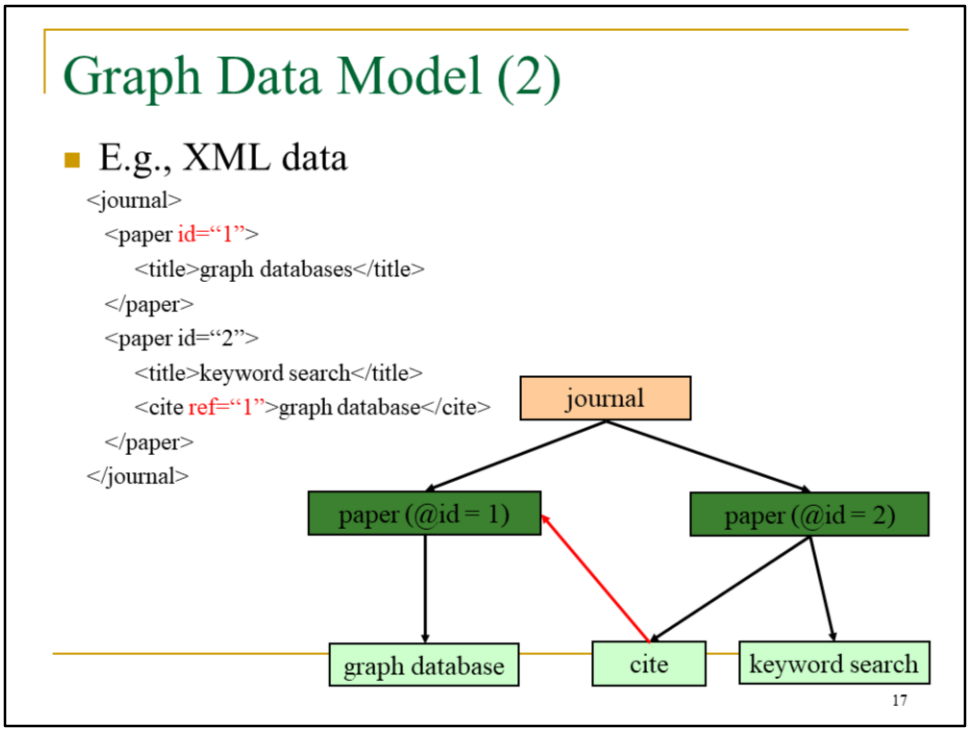


The keyword search was proposed to query data such as relational, XML, HTML data. In general, the keyword search query retrieves a closely connected set of nodes that match all given query keywords together.

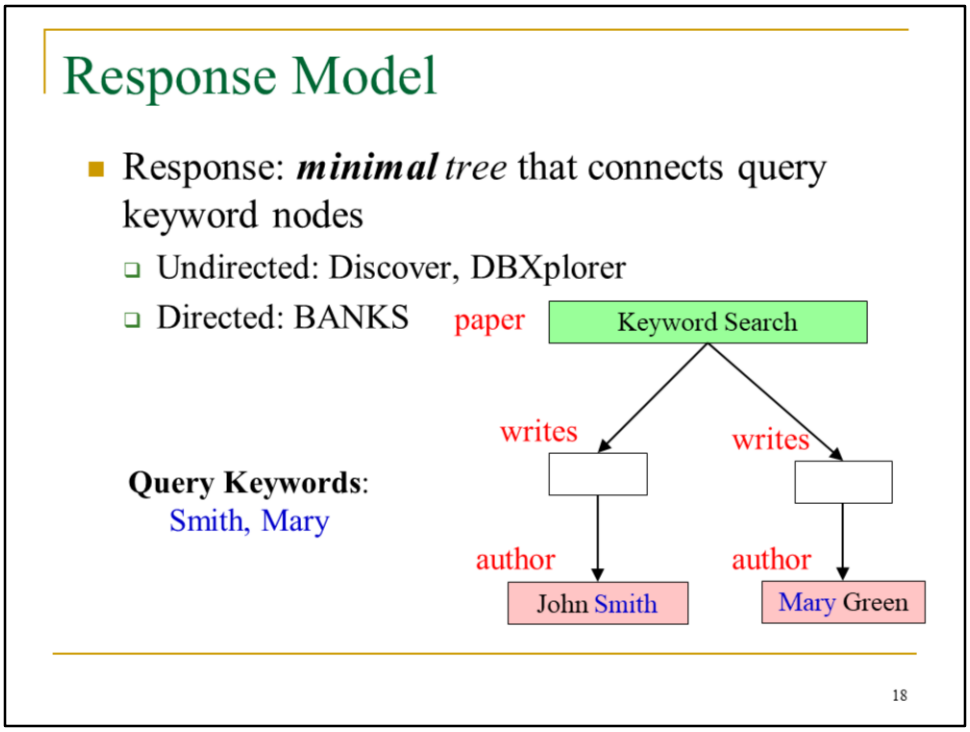
In the literature, previous techniques include BANKS, Discover, DBXplorer, Xrank, BLINKS, and so on. The main focus of these works is to design efficient search algorithms to find connections among nodes.



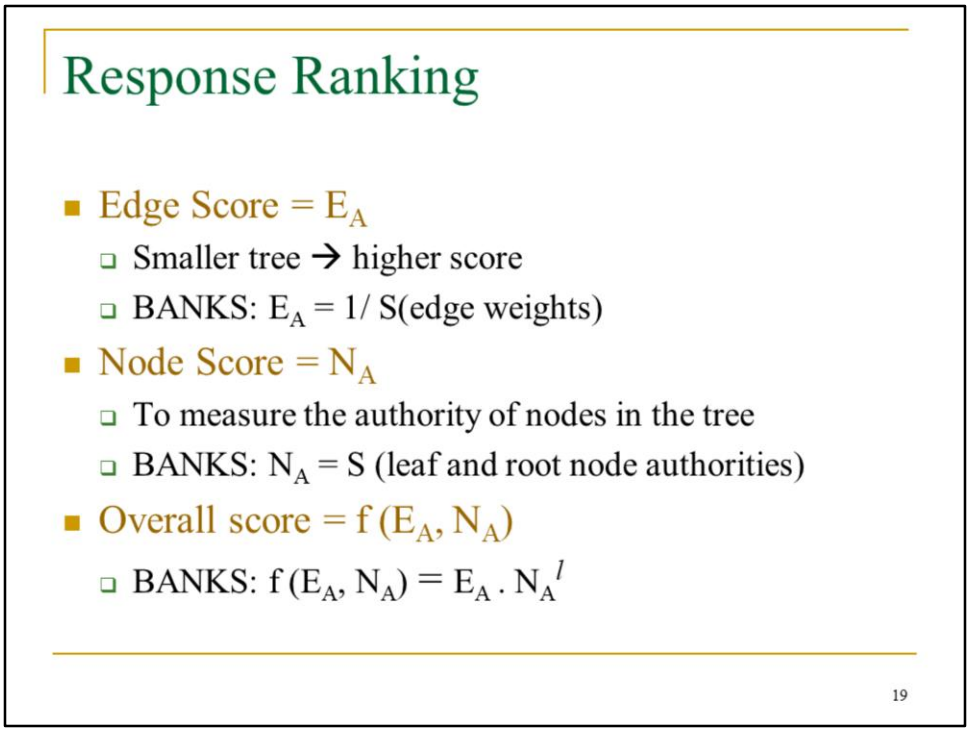
Previous works usually model the underlying data as graphs, for example, a directed weighted graph for the BANKS approach. Taking DBLP data as an example, each node in the graph can be a tuple (e.g., a paper or an author), whereas each edge represents the foreign key references. As shown in the figure, a paper may be written by multiple authors, and thus a paper can be pointed to by multiple author entities, where each author has "writes" relationship with this paper. This way, the DBLP data can be represented by a directed graph.



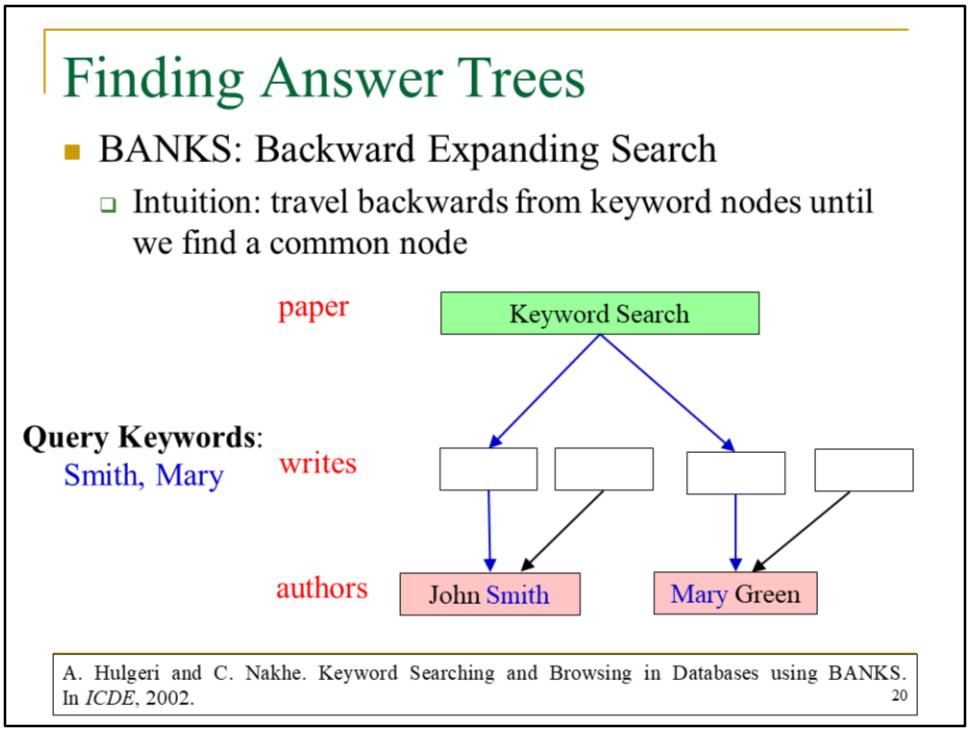
XML data are naturally represented by trees (or a special case of graphs). As shown in the figure, under the root "journal", there are two children, that is, papers with IDs 1 and 2. Paper 1 has a child title "graph database", and Paper 2 has two children, a title "keyword search" and a "cite" relationship with Paper ID 1 (which is a cross reference). The XML documents can be usually represented by tree structures (without any circles). However, in the presence of cross references between siblings or any node pairs, circles may exist, and thus they are represented by graphs.



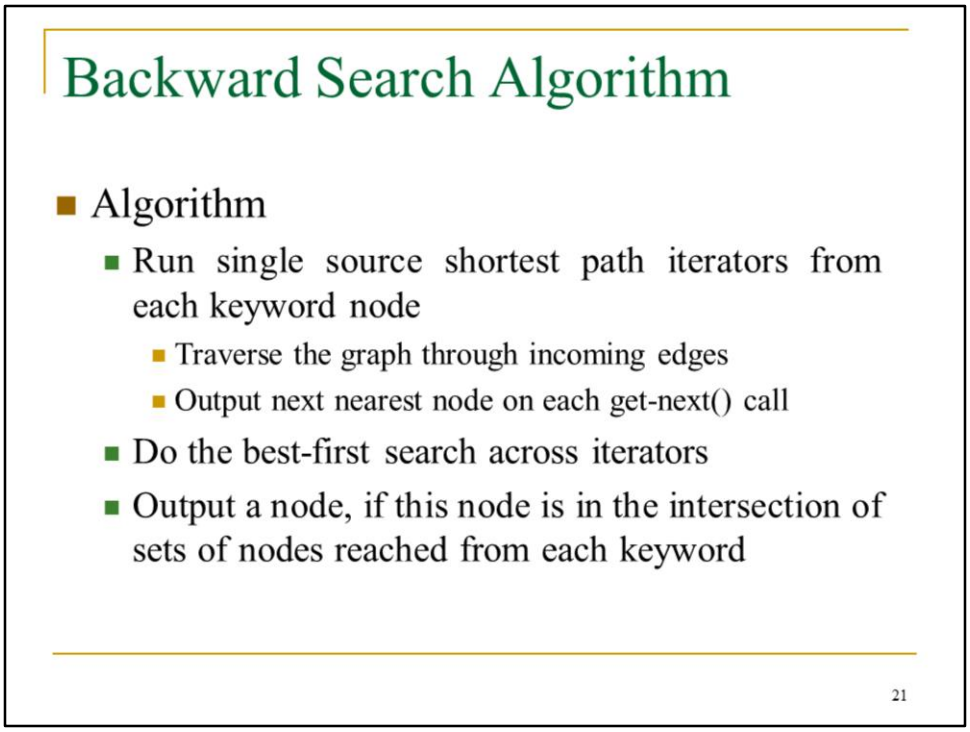
For the keyword search, we usually want to find a minimal subgraph that involves the user-specified keywords. One example is a rooted tree connecting nodes with the given query keywords. As shown in the figure, given two query keywords, Smith and Mary, we can see that these two keywords appear in two author nodes “John Smith” and “Mary Green”, respectively. The keyword search algorithm will find one minimum tree structure, with paper “Keyword Search” as the root that connects to these two keyword nodes.



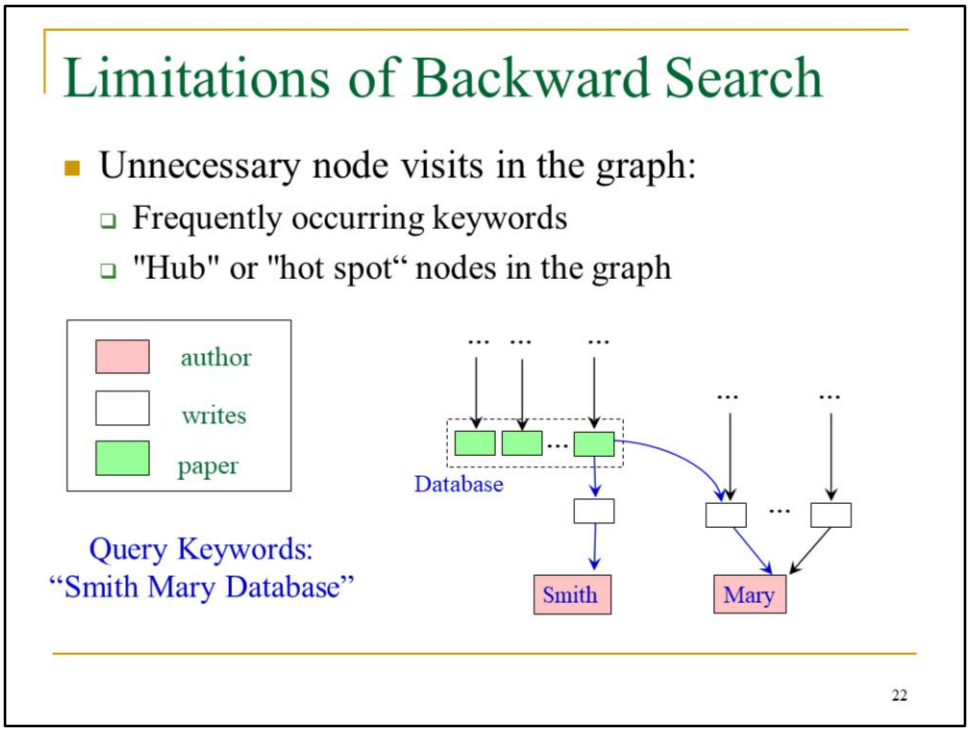
Here, the minimal tree structure can be defined as the tree with the highest score, with respect to edge and node scores. For BANKS approach, the edge score can be defined by 1 over edge weights, and the node score can be given by a function of authorities of nodes in the tree. The overall score is given by a function w.r.t. these two scores, for example, in BANKS, it is defined as edge score times the node score to the power of *I*.



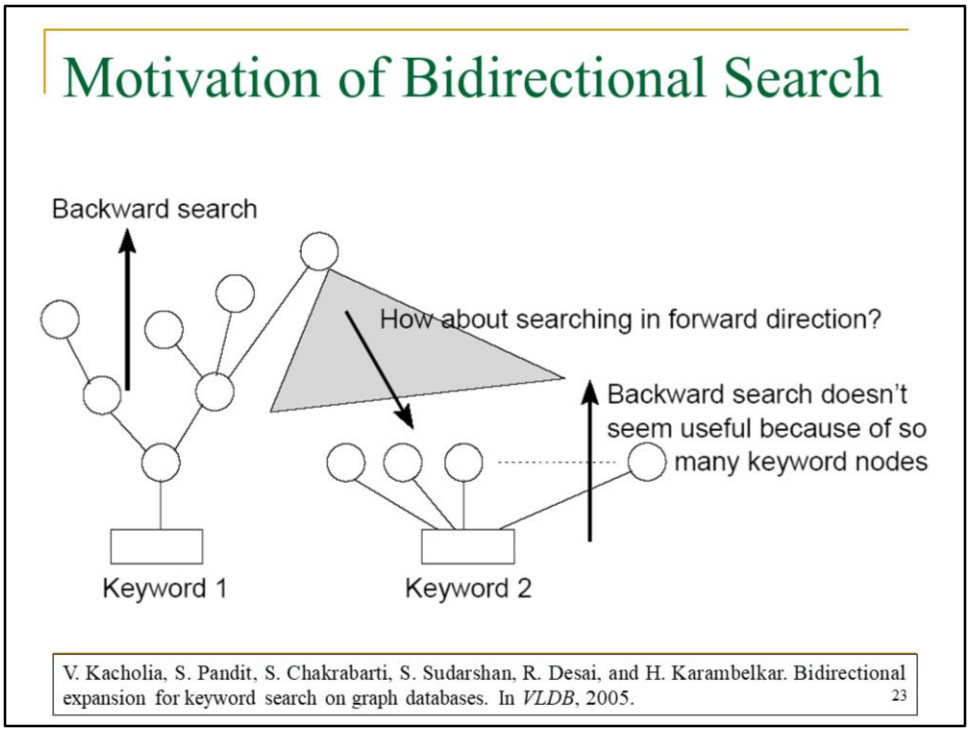
The BANKS keyword search approach uses a backward expanding search strategy. Intuitively, it starts from those keyword nodes and traverse backwards until we find a common node, which is a root. As shown in the example of the figure, we start from 2 author nodes containing keywords “Smith” and “Mary”, respectively, and traverse the graphs through incoming edges, until we find a common node, paper node with title “Keyword Search”.



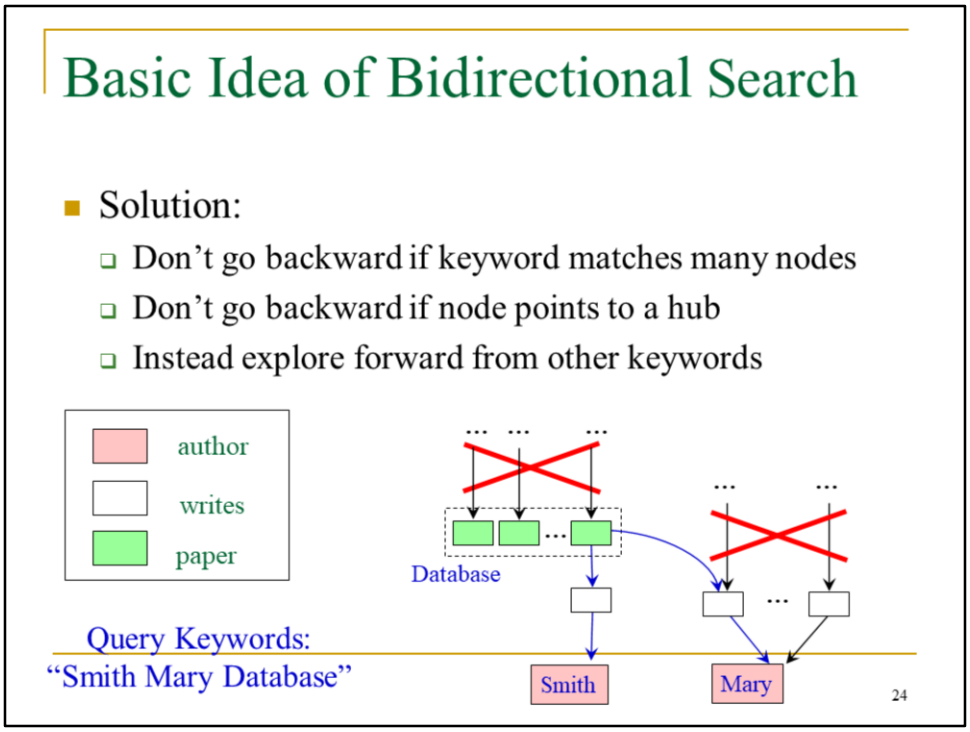
Here is the pseudo code of the backward search algorithm.



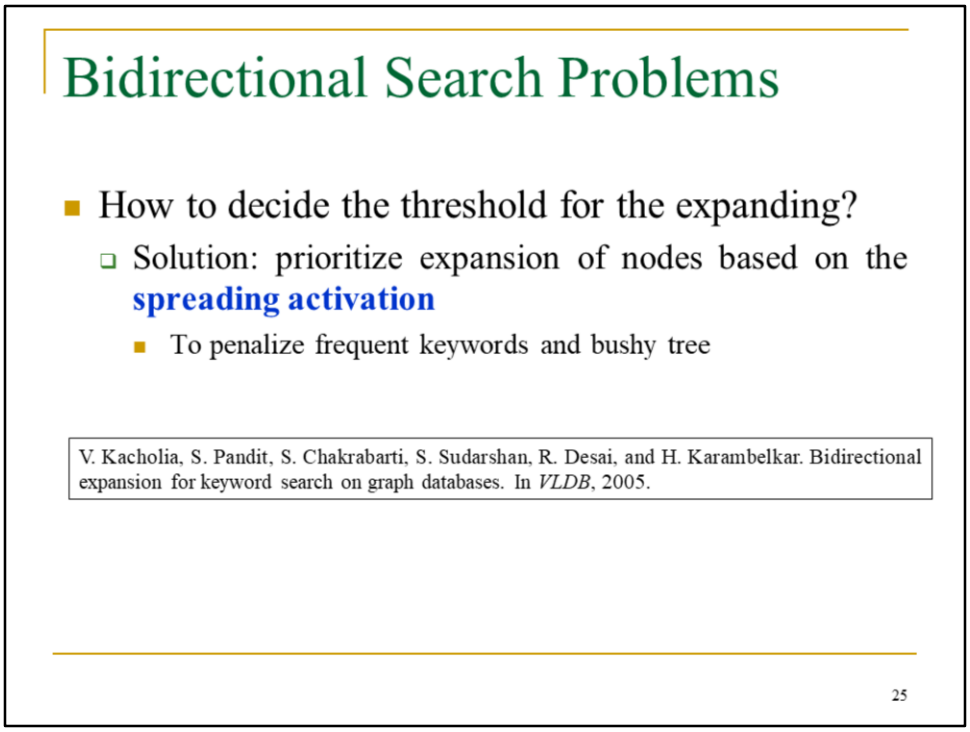
However, there are some limitations for the backward search algorithm. If the query keywords are frequently occurring in the graph, we may have to access many unnecessary query nodes. As an example in the figure, although there is only one answer tree for the keyword search (with keywords “Smith Mary Database”), we have to visit many nodes containing the keyword “Database”. Similarly, when a keyword node (e.g., “Mary” in the figure) has many incoming edges (we call it hub or hot spot node in the graph with high in-degree), in this case, we also have to access many unnecessary nodes through incoming edges.



To tackle the problem encountered in the backward search, bidirectional search approach was proposed. The basic idea is to combine backward search with forward search at the same time.

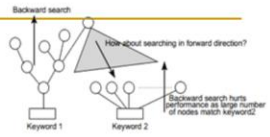


Intuitively, bidirectional search switches to the forward search, when the backward search encounters many nodes with query keywords, or encounters hub nodes with high in-degrees.



One interesting problem is that how we can decide if we should stop backward search and switch to forward search. The solution is to prioritize expansion of nodes based on the spreading activation, by penalizing those frequent keywords and bushy trees.

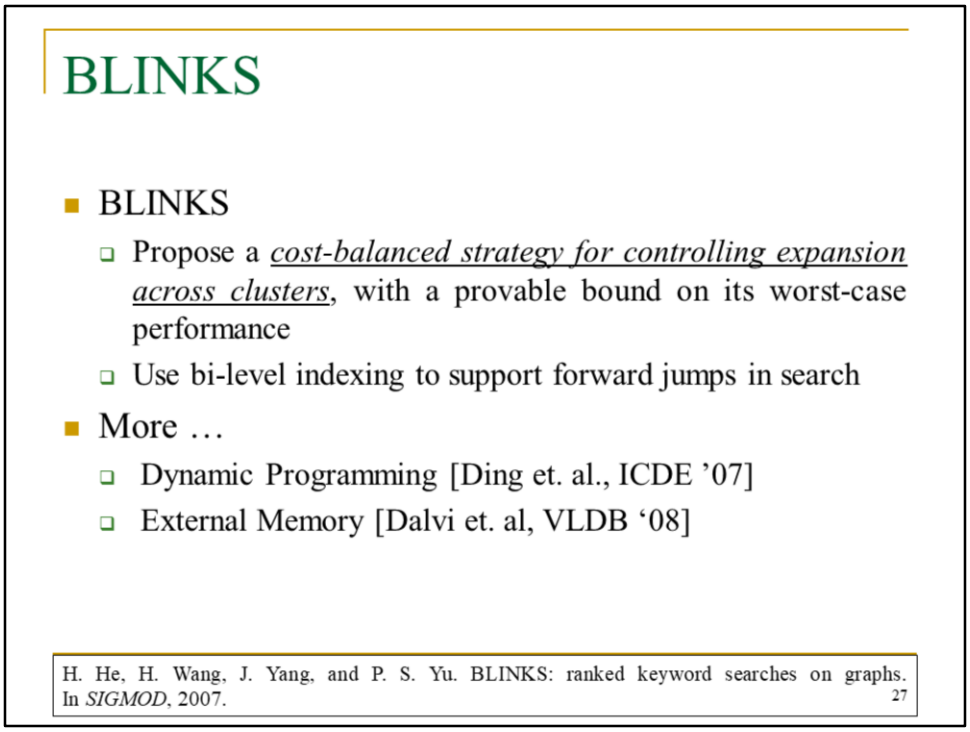
Bidirectional Search



- Backward Search: Merge all the single source shortest path iterators from the Backward search algorithm into a single iterator, called the *incoming iterator*
- Forward Search: Concurrently run an *outgoing iterator*, which follows forward edges starting from *all the nodes* explored by the incoming iterator
- Prioritize: Spreading activation to prioritize the search, which chooses incoming iterator or outgoing iterator to be called next

26

Here are some more details about the bidirectional search. For the backward search, an incoming iterator will merge all single source shortest path iterators from Backward search algorithm. The forward search is handled by an outgoing iterator, which follows forward edges starting from all nodes explored by the incoming iterator. To enable the bidirectional search, the next searching mode is decided by spreading activation, which chooses which iterator (incoming or outgoing) to be called next.



Another BLINKS approach retrieves top-k answer trees from the graph with the highest ranking scores. It proposes a cost-balanced strategy for controlling the expansion across clusters, and uses a bi-level index to support efficient exploration of the graph.

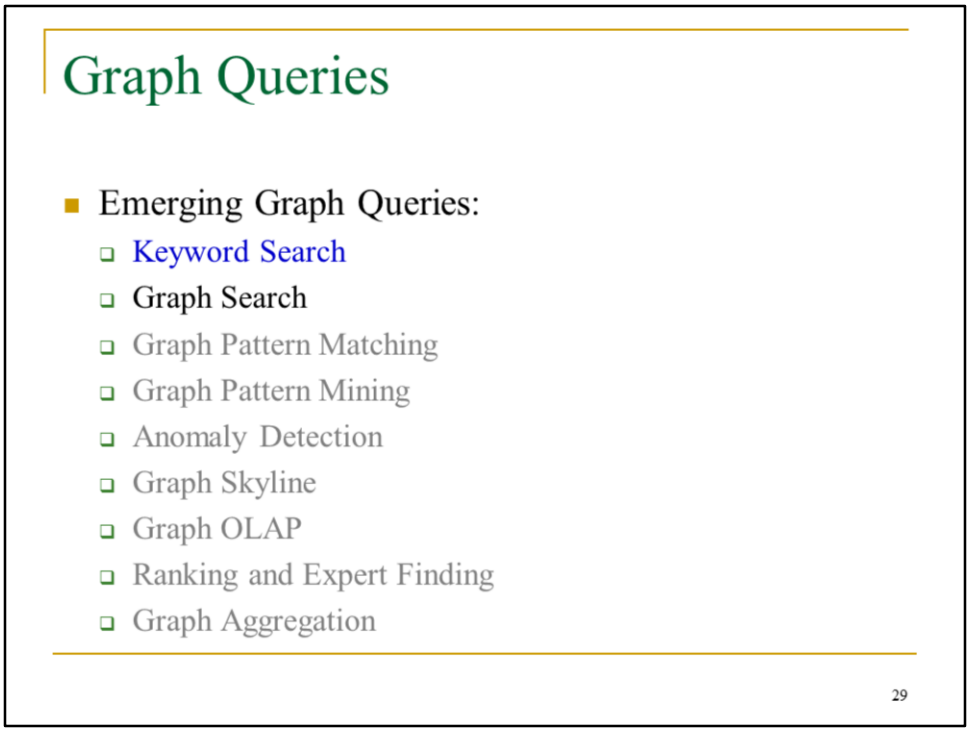
There are some other papers about BLINKS. If you are interested in more details, you can read these papers.

Outline

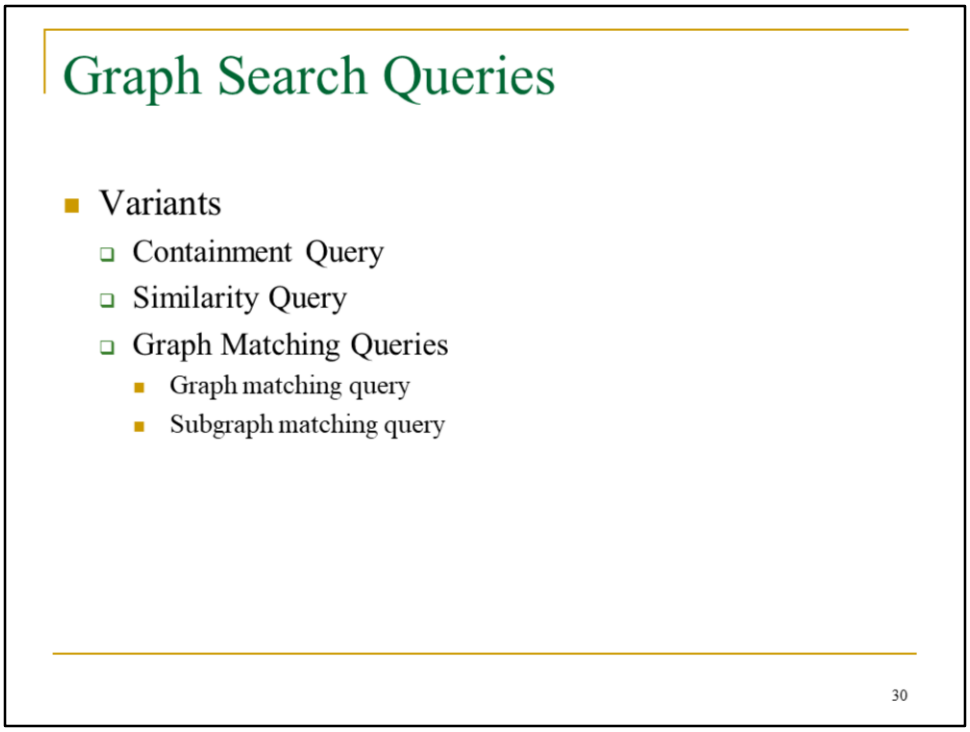
- Aggregate Queries
- Keyword Search Query
- Graph Queries

28

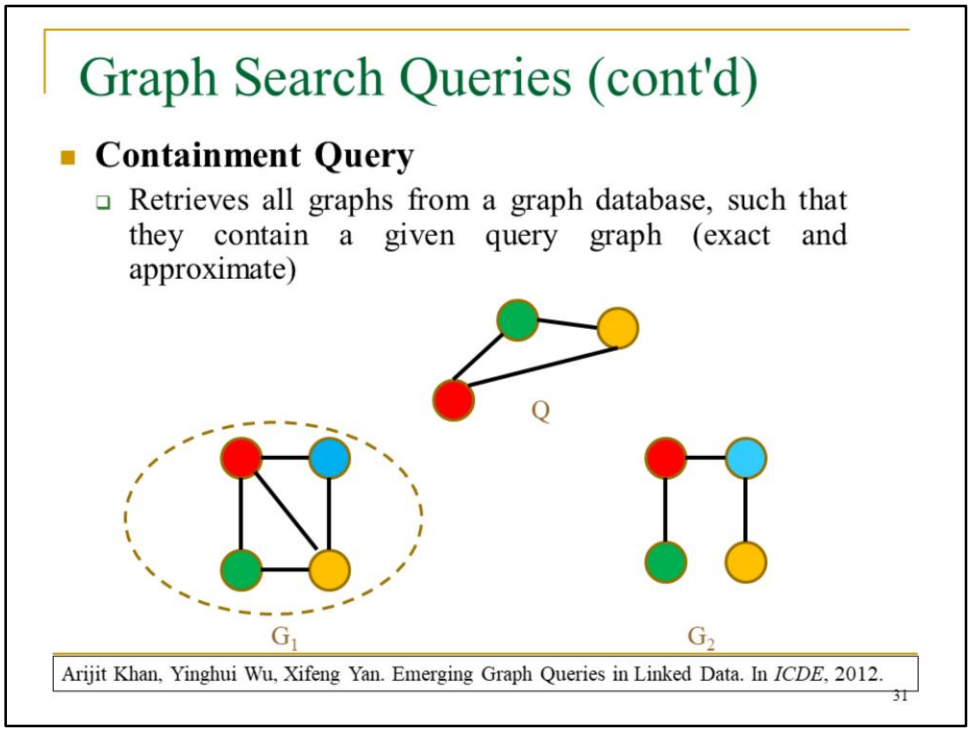
Next, we will discuss graph queries.



There are many graph queries. The keyword search in graphs is one of them. There are also some classical queries in graph databases.

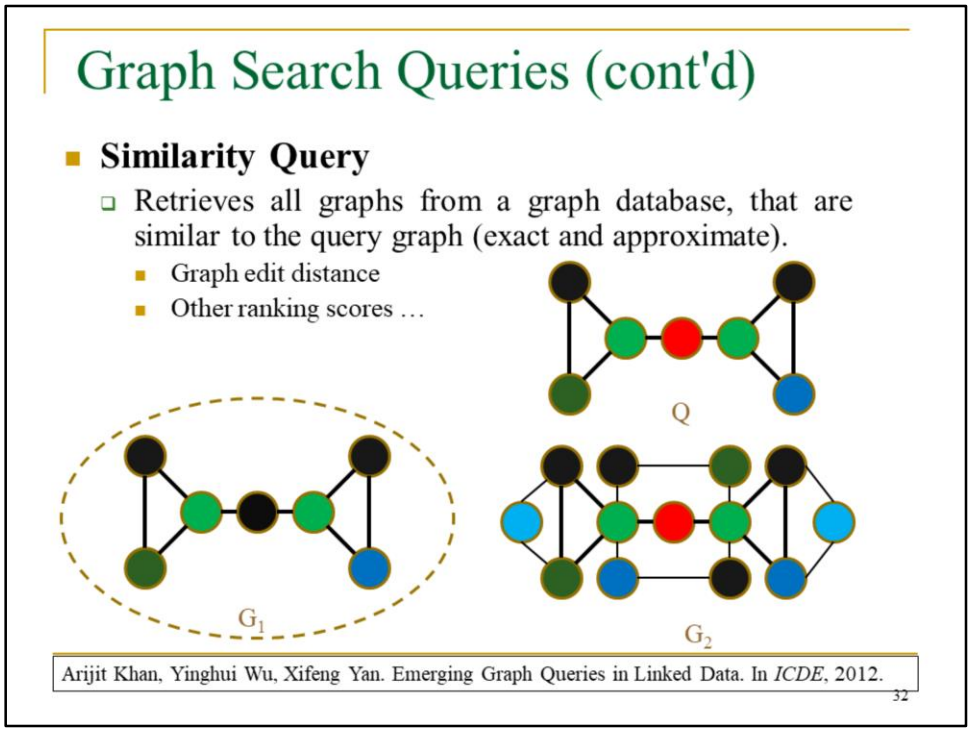


For example, the graph search queries include, but are not limited to, containment query, similarity query, and graph matching queries. Graph matching queries can be classified into graph and subgraph matching queries.



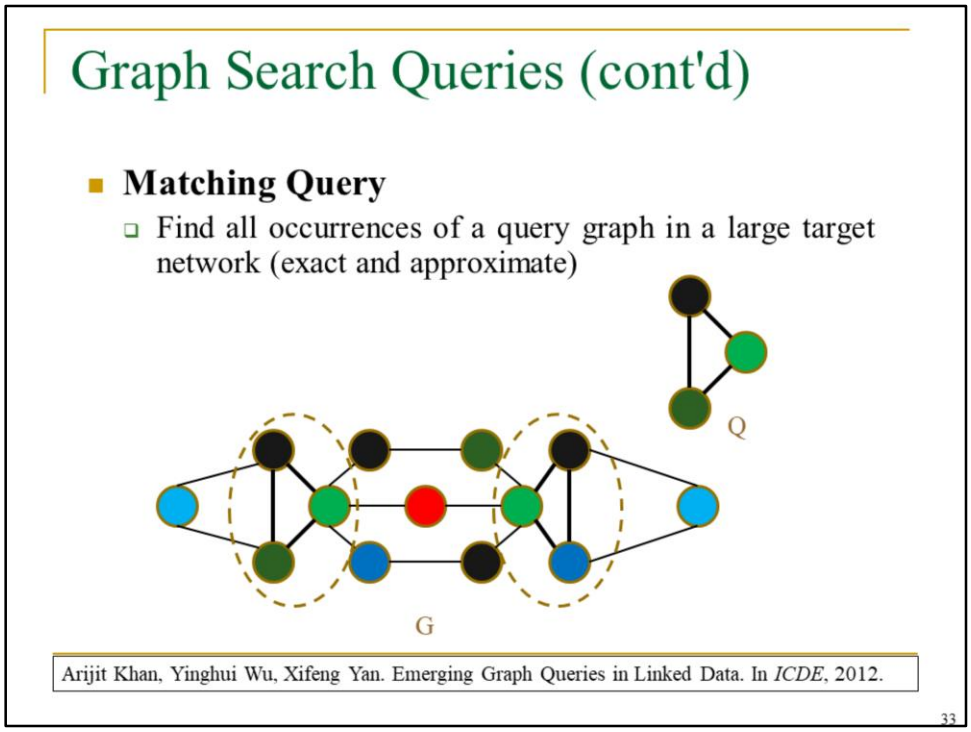
Given a graph database and a query graph, a containment query retrieves all graphs from the database that contain the query graph. The returned graph answers can be either exact or approximate.

As shown in the example of the figure, the graph database contains 2 data graphs, G_1 and G_2, and the query graph is Q. In this example, G_1 will be the answer to the containment query, since Q is a subgraph of data graph G_1.



For the similarity query, given a graph database and a query graph, we want to retrieve all graphs from the database that exactly or approximately match with the query graph. Here, two graphs match with each other, if they are similar with measures such as the graph edit distance or other ranking score functions.

In the figure, we can see that data graph G_1 approximately matches with the query graph Q, since there is one node difference (i.e., with different colors/labels).



For the matching queries, we want to exactly or approximately find all occurrences of a query graph in a large-scale graph. As shown in the figure, there are two subgraphs in data graph G that are structurally isomorphic to the query graph (but with some color/label difference). This type of queries has the real applications in large graphs such as social networks or biological networks.