Advanced Databases: Exam reference solution

Pierre Senellart

14 December 2022

A first version of this reference solution has been produced by GPT-4, which has then been manually verified, corrected, modified. Because of this, answers tend to be long and verbose – this is suitable for a reference solution, but it is not expected that students write such detailed answers. Much more concise answers are fine. For many questions, there is not one valid answer: there are many ways to approach the problem.

1. (1 point)

   To estimate the amount of data required to store 50 years of TV programs, we can make the following assumptions:

   - There are 100 TV channels.
   - Each channel schedules 20 programs per day, on average, for 365 days per year (ignoring leap years), for a total of 7 300 programs per year per channel.
   - Each program has meta-information that takes an average of 10 000 bytes to store.
   - Each hour of archived video, including audio and subtitle tracks, takes 10^9 bytes to store.

   Based on these assumptions, we can estimate the amount of data required to store 50 years of TV programs:

   Meta-information: 100 channels $\times$ 7 300 programs/year $\times$ 10 000 bytes/program $\times$ 50 years = 3.65×10^{11} bytes, or 365 gigabytes.

   Video information: 100 channels $\times$ 7 300 programs/year $\times$ 1 hour/program $\times$ 10^9 bytes/hour $\times$ 50 years = 3.65×10^{16} bytes, or 36.5 petabytes.

   To store and manage 365 gigabytes of meta-information, a relatively small storage system could be used, such as a single high-capacity hard drive or a cloud-based storage service. The data could also be backed up using a redundant array of independent disks (RAID) system or a cloud-based backup service.

   However, storing and managing 36.5 petabytes of video information would require a much more complex and expensive storage infrastructure. A high-capacity storage system, such as a network-attached storage (NAS) or a storage area network (SAN), would be needed. The system would likely consist of many high-capacity hard drives connected to a large amount of nodes (of the order of 1000), configured in a RAID system for data redundancy and protection against disk failure. Additionally, a backup system would be necessary, such as an off-site backup or a cloud-based backup service, to ensure data recovery in case of disasters.

   Given the large amount of data involved, the hardware required to store and manage it would be expensive and require careful planning and maintenance to ensure its reliability and longevity. The archival institution would need to consider factors such as power consumption, cooling, and physical space requirements when designing the storage system.

2. (3 points)

   Here is a proposed schema for storing all meta-information about the TV programs:

   a) Programs
      - id (integer, primary key)
      - title (text)
      - type (text)
      - summary (text)
      - duration (integer)

   b) Persons
      - id (integer, primary key)
      - name (text)

   c) ProgramPersons
      - program_id (integer, foreign key to Programs)
      - person_id (integer, foreign key to Persons)
      - role (text)

   d) Series
      - id (integer, primary key)
      - title (text)
      - description (text)
      - type (text)

   e) Episodes
      - id (integer, primary key, foreign key to Programs)
      - series_id (integer, foreign key to Series)
      - season_number (integer)
      - episode_number (integer)

   f) ProgramSchedulings
      - program_id (integer, foreign key to Programs)
      - tv_channel (text)
      - start_datetime (datetime)
      - end_datetime (datetime)

   This schema allows for the storage of information about individual programs, the persons involved in their creation, series, episodes, and schedulings. Each table has a primary key column (id) that uniquely identifies each row, and foreign key constraints are used to link the tables together.

   Storing this amount of meta-information is feasible in a regular database management system such as Oracle or PostgreSQL, as the total amount of data is relatively small (365 gigabytes). The schema is designed to be efficient and scalable, allowing for easy retrieval of information and the addition of new data as needed.

   However, storing the video information is a much more challenging task due to its large size (36.5 petabytes). It would likely require a specialized storage system and data management approach, such as a distributed file system or a cloud-based storage solution, rather than a traditional database management system. The video files could be stored on separate storage devices and linked to the meta-information using a unique identifier or filename.

3. (1 point)

   Assuming that the Fort Boyard game show is part of a series in the Episodes table, and that the ProgramPersons table links programs to persons involved:

```
SELECT person_id, name, MIN(start_datetime) AS earliest_airdate
FROM ProgramPersons
JOIN Persons ON person_id = Persons.id
JOIN Programs ON program_id = Programs.id
JOIN Episodes ON Programs.id = Episodes.id
JOIN ProgramSchedulings ON Programs.id = ProgramSchedulings.program_id
WHERE Episodes.title = 'Fort Boyard'
GROUP BY person_id, name
```

   This query retrieves the roles and names of all participants involved in the Fort Boyard game show, as well as the earliest airdate for each episode they participated in. It does this by joining the relevant tables together and filtering the results based on the Fort Boyard episode title. The MIN() function is used to get the earliest airdate for each episode.

4. (1 point)

   The number of updates that would typically be issued on a database containing information about TV programs would depend on the specific use case and the level of activity of the institution. However, in general, it is likely that updates would be infrequent compared to the amount of read operations, as the majority of the data is historical and unlikely to change. The updates could include new program information, changes to scheduling information, and updates to the participants or creators of programs. The order of magnitude of the number of updates would be 2 000 per day, as this is the number of programmings per day – this represents a very low bandwidth of updates.

   A storage solution that implements strong ACID guarantees may not be strictly required for such an application, as the amount of updates is likely to be relatively low. However, ACID guarantees can provide important benefits in terms of data consistency, durability, and isolation, which can be critical for an archival institution. For example, if the institution needs to ensure that all data is accurately recorded and that updates are made in a safe and consistent way, then a storage solution that provides strong ACID guarantees may be necessary.

   Additionally, if the institution needs to perform complex queries or generate reports that involve aggregating or joining data from multiple tables, ACID guarantees can help ensure that the results are accurate and consistent. Finally, if the institution is subject to regulations or requirements that mandate strong data consistency and durability, a storage solution that implements ACID guarantees may be necessary to meet those requirements.

5. (1 point)

   When using BigTable/HBase to store the video information, one way to organize the data into the HBase data model is as follows:

   Key: A unique identifier for each video, typically the program identifier as stored in the meta-information database.

   Column families: One column family for the video file, one for soundtracks and one for subtitles.

   Columns: A column for each video file format and a column for each language or subtitle tracks

For example, the HBase table could be named "VideoData" and have the following column families:

- `"Video"`: contains columns for the original video file and any other video formats, such as compressed versions or versions in different resolutions. `"Audio"`: contains columns for each soundtrack language, with each column storing the audio data in that language. `"Subtitles"`: contains columns for each subtitle language, with each column storing the subtitle data in that language.

To determine a reasonable number of computers/nodes in a BigTable cluster storing this data, we need to consider the amount of data to be stored and the expected read and write throughput. Given that we need to store 36.5 petabytes of data and that video files are typically large, we would need a large number of nodes to achieve reasonable performance and redundancy. A reasonable estimate could be in the range of 100-10000 nodes, depending on the specific requirements and performance goals of the system. However, the exact number of nodes required would need to be determined through testing and benchmarking with the actual data and workload.

6. (1 point)

   When using a key-value store such as a distributed hash table (DHT) to store the video information, one way to organize the data is as follows:

   Key: A unique identifier for each video, typically the program identifier as stored in the meta-information database.

   Value: A data structure containing information about the video file, including the video data and any soundtrack and subtitle tracks.

   Alternatively, the key could be composite, with the program identifier and a part corresponding to the different parts of the video file: video, subtitle or audio track for a specific language.

   The specific format of the value would depend on the requirements and capabilities of the DHT system used. For example, if the DHT supports storing large values, the value could contain the entire video file and associated tracks. Alternatively, if the DHT has size limitations on the value, the value could contain references to the actual video file stored in a separate storage system.

   To determine a reasonable number of nodes in a DHT cluster storing this data, we need to consider the amount of data to be stored, the expected read and write throughput, and the requirements for redundancy and fault tolerance. A reasonable estimate could be in the range of 100-10000 nodes, depending on the specific requirements and performance goals of the system. However, the exact number of nodes required would need to be determined through testing and benchmarking with the actual data and workload. Additionally, the DHT system would need to be designed with replication and consistency mechanisms to ensure data durability and consistency in the face of node failures and network partitions.

7. (1.5 point)

   Here is an example MapReduce program (in pseudo-code) that computes representative images (thumbnails) for every video stored in HBase, and stores the result back into HBase:

```
Mapper:
Input: video key-value pairs from HBase
  (key = unique identifier for the video, value = video file data)
Output: video key-thumbnail value pairs
```

```
  (key = same as input, value = thumbnail image data)

def map(key, value):
    # Load video data
    video_data = load_video_data(value)
    # Compute thumbnail image
    thumbnail = compute_thumbnail(video_data)
    # Output key-thumbnail pair
    yield key, thumbnail

Reducer:
Input: video key-thumbnail value pairs from Mapper
  (key = same as input, value = thumbnail image data)
Output: video key-thumbnail value pairs to HBase
  (key = same as input, value = thumbnail image data)

def reduce(key, values):
    # Store thumbnail image data in HBase
    store_thumbnail_in_hbase(key, values[0])
    # Output key-thumbnail pair
    yield key, values[0]

Main program:
# Set up HBase input and output
hbase_input = create_hbase_input()
hbase_output = create_hbase_output()

# Set up MapReduce job
job = create_mapreduce_job(hbase_input, hbase_output)
job.setMapperClass(Mapper)
job.setReducerClass(Reducer)
job.setOutputKeyClass(Text)
job.setOutputValueClass(BytesWritable)

# Run MapReduce job
job.waitForCompletion(true)
```

The program uses the HBase input format to read video data from HBase and the HBase output format to store thumbnail data back into HBase. The Mapper loads each video file, computes a thumbnail image, and outputs the key-value pair with the video key and thumbnail data. The Reducer stores the thumbnail data back into HBase and outputs the key-value pair. The Main program sets up the MapReduce job and runs it.

Note that the functions `load_video_data`, `compute_thumbnail`, and `store_thumbnail_in_hbase` are not provided here and would need to be implemented based on the specifics of the video data and thumbnail generation requirements.

An important observation is that for such an application, the reducer does very little: this is an *embarassingly parallel* task, that does not require any form of aggregation.

8. (4 points)

RDF and SPARQL To represent meta-information about a specific episode of a TV drama using RDF, we could use the following ontology:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .
@prefix tvd: <http://example.com/tv-drama#> .

tvd:Episode123 a schema:TVEpisode ;
    schema:name "Episode 123" ;
    schema:partOfSeason tvd:Season2 ;
    schema:partOfSeries tvd:TVSeries1 ;
    schema:datePublished "2002-04-01"^^xsd:date ;
    schema:duration "PT45M"^^xsd:duration ;
    schema:description "Description of Episode 123" ;
    schema:actor tvd:Actor1, tvd:Actor2 ;
    schema:director tvd:Director1 ;
    schema:producer tvd:Producer1 ;
    schema:publication tvd:Broadcast1 ;
    rdf:type tvd:ComedyShow .

tvd:Season2 a schema:TVSeason ;
    schema:name "Season 2" ;
    schema:partOfSeries tvd:TVSeries1 .

tvd:TVSeries1 a schema:TVSeries ;
    schema:name "TV Series 1" ;
    schema:description "Description of TV Series 1" .
```

In this example, we have an Episode resource with properties such as its name, duration, description, actors, director, producer, and the date it was published. It is also part of a Season and a TV Series.

To retrieve all different types of programs scheduled on January 1st, 2000, at 00:00, using SPARQL, we could use the following query:

```
PREFIX schema: <http://schema.org/>
PREFIX tvd: <http://example.com/tv-drama#>

SELECT DISTINCT ?type
WHERE {
  ?program schema:datePublished "2000-01-01T00:00:00"^^xsd:dateTime ;
           rdf:type ?type .
}
```

This query selects all programs that have a publication date of January 1st, 2000, at 00:00, and retrieves their RDF type. The DISTINCT keyword ensures that only unique types are returned.

XML and XQuery To represent meta-information about a specific episode of a TV drama using XML, we could use the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<tv-drama xmlns="http://example.com/tv-drama">
```

```
  <episode type='ComedyShow'>
    <name>Episode 123</name>
    <season>Season 2</season>
    <series>TV Series 1</series>
    <datePublished>2002-04-01</datePublished>
    <duration>PT45M</duration>
    <description>Description of Episode 123</description>
    <actors>
      <actor>Actor 1</actor>
      <actor>Actor 2</actor>
    </actors>
    <director>Director 1</director>
    <producer>Producer 1</producer>
    <publication>
      <broadcast>Channel 1</broadcast>
      <start>2002-04-01T19:00:00</start>
      <end>2002-04-01T19:45:00</end>
    </publication>
  </episode>
  <season>
    <name>Season 2</name>
    <series>TV Series 1</series>
  </season>
  <series>
    <name>TV Series 1</name>
    <description>Description of TV Series 1</description>
  </series>
</tv-drama>
```

In this example, we have an episode element with sub-elements such as its name, duration, description, actors, director, producer, and the date it was published. It is also part of a season and a TV series. The publication element contains information about the TV channel, start, and end times.

To retrieve all different types of programs scheduled on January 1st, 2000, at 00:00, using XQuery, we could use the following query:

```
declare namespace td = "http://example.com/tv-drama";

distinct-values(
  for $program in
    //td:publication[td:start = xs:dateTime('2000-01-01T00:00:00')]
  return $program/parent::td:episode/@type
)
```

This query selects all publication elements that have a start time of January 1st, 2000, at 00:00, and returns the unique values of their parent episode's type attribute. The distinct-values() function ensures that only unique types are returned.

9. (1.5 point)

   For the specific use case of archival of data and metadata about TV programmings, each of the three technologies has specific advantages and disadvantages.

Advantages of RDF+SPARQL:

- RDF is well-suited for representing metadata, which can be highly diverse and unstructured, allowing different metadata fields to be easily added as needed.
- RDF's flexible schema-less data model makes it well-suited for handling evolving metadata over time, as new fields can be easily added or modified.
- SPARQL's graph querying capabilities are well-suited for querying complex relationships between TV programs, their schedules, and their associated metadata.

Disadvantages of RDF+SPARQL:

- RDF can be less efficient for storing and indexing large amounts of metadata compared to more optimized data structures, such as columnar stores.
- SPARQL queries can be complex and difficult to optimize for large datasets, requiring specialized query planning and indexing techniques.
- The lack of standardization in RDF vocabularies and ontologies can make it difficult to integrate metadata from different sources.

Advantages of XML+XQuery:

- XML's hierarchical structure makes it well-suited for representing metadata about TV programs, which can have complex and nested relationships.
- XQuery provides a powerful querying language for hierarchical data, which can be well-suited for querying metadata about TV programs.
- XML has been widely used in the media industry for storing metadata about TV programs, making it a familiar and established technology in this domain.

Disadvantages of XML+XQuery:

- XML can be verbose and inefficient for storing large amounts of metadata, which may require specialized optimization techniques, such as XML compression.
- XQuery may not be as well-supported by third-party tools and technologies compared to other querying languages, such as SQL or SPARQL.
- XML's hierarchical structure may not be well-suited for representing complex relationships between TV programs, such as social media interactions or viewer engagement.

Advantages of Graph Databases+Cypher:

- Graph databases are well-suited for representing complex relationships between TV programs, their schedules, and their associated metadata.
- Cypher provides a user-friendly querying language that can be well-suited for non-experts, such as archivists or media analysts.
- Graph databases can be highly performant for querying complex relationships between TV programs and their metadata, which can be useful for analytics or search applications.

Disadvantages of Graph Databases+Cypher:

- Graph databases may require specialized indexing and storage mechanisms to optimize querying performance, which may require more expertise compared to traditional relational databases.
- Graph databases may not be well-suited for representing structured data, such as tabular data with many rows and columns, which may be a limitation for certain use cases.

- Graph databases may not be as widely adopted in the media industry compared to other technologies, which may limit the availability of expertise and third-party tools.

10. (1 point)

    Ontology-based querying involves using a formal ontology to describe and query data. In this situation, the ontology would define a taxonomy of TV program types, including the specific characteristics that define each type, such as the topic or theme of the program. An ontology-based query would then use this ontology to identify TV programs that have specific characteristics, such as a particular topic or theme.

    Imagine for instance that there is a type "Comedy" that encompasses subconcepts in the ontology such as "DarkComedy" or "SitCom". When a user queries for a "Comedy" drama, she should be able to obtain results that are marked as "DarkComedy", as the ontology specificies "DarkComedy" is a subtype of "Comedy". This requires ontology-based querying, as disregarding the ontology would not produce the expected result.

11. (2 points)

    To design an architecture for cataloging and archiving TV programs over 50 years, I would propose a combination of technologies for storing and querying both the meta-information and the video information.

    For the meta-information, I would use RDF+SPARQL. RDF+SPARQL would allow for flexible and schema-less data modeling, making it easy to add new metadata fields as needed and integrate metadata from different sources. SPARQL's graph querying capabilities would also be useful for querying complex relationships between TV programs, their schedules, and their associated metadata.

    For the video information, I would use a distributed storage solution such as Hadoop/HDFS or Amazon S3, as well as a distributed computing framework such as Apache Spark to process and analyze the video data. Hadoop/HDFS or Amazon S3 would allow for scalable and cost-effective storage of large amounts of video data, while Spark would provide a distributed computing framework for processing and analyzing the video data at scale. I would also consider using specialized video processing libraries and tools, such as FFmpeg or OpenCV, for extracting metadata and generating representative images or thumbnails of the videos.

    Overall, this architecture would allow for flexible and scalable storage and querying of both the meta-information and video information, using a combination of established and widely adopted technologies that are well-suited for the specific requirements of cataloging and archiving TV programs over 50 years.

12. (2 points)

    An approach to identifying the original source of the data is to use data provenance. Data provenance is a technique for tracking the complete history of how data is created, modified, and propagated through a system. In this case, data provenance could be used to track the original source of the data and any subsequent modifications that were made to it.

    To implement data provenance in this use case, each time a TV production company or TV channel enters data into the database, a provenance record would be created that captures the source of the data, the timestamp of the entry, and any other relevant metadata about the data entry. Each time the data is subsequently modified or updated, a new provenance record would be created that captures the changes made, along with the identity of the person who made the changes and the timestamp of the update.

When issuing a query on the meta-information, users could access the provenance records associated with each metadata record, allowing them to trace the complete history of how the data was created and modified. Users could also use provenance to identify the original source of the data and any subsequent modifications made to it. For example, a query for all program schedulings could include a filter criterion for the provenance record, allowing users to restrict the results to schedulings that were entered by a specific TV channel or TV production company.

Obtaining provenance for the result of a complex query involves tracing the sources of the data used in the query and the operations performed on that data during the query execution. In other words, we need to track the lineage of the data used in the query, from its original sources through any intermediate steps to the final output.

One approach to obtaining provenance for a complex query is to use a provenance tracking system that automatically captures the lineage of data as it moves through the system. For example, the system could track each data item and its associated metadata, including its source, transformations applied to it, and the outputs produced by those transformations. The system could then use this lineage information to generate a complete provenance record for each query result.