Q: Can we able to save and efficiently retrieve shows / movies viewed by user in sequence (say, shows viewed in a given session) in the order of watching?

A: this can be best done by storing the current timestamp whenever there is a “watch” event. For example, when a user watches a session in a session, store it like this (example using JavaScript):

```
db.watched.insert({
  userId: ...,
  movieId: ...,
  timestamp: Date.now()
});
```

Then there is a timestamp available in attribute “timestamp”.
If there is a sorted index on ["userId", "timestamp"], this allows efficiently retrieving the watched movies of a user sorted by time:

```
db.watched.ensureIndex({
  type: "skiplist",
  fields: ["userId", "timestamp"]
});
```

The retrieval query utilizing this index then is:

```
FOR doc IN watched
  FILTER doc.userId == @userId
  SORT doc.timestamp
  RETURN doc
```

If the above should be done by session, then also add a “sessionId” attribute to track and filter the session id. That attribute then should also be indexed.

Q: Dynamic queries

A: AQL queries are passed to the server as a string.
A client application can construct a query using simple string concatenation, e.g.

```
var query = "FOR doc IN " + someCollection +
            " FILTER doc." + field + " == " + userSuppliedValue +
            " RETURN doc";
```

Obviously this is not safe, as attackers may inject arbitrary query fragments.

A better way to construct queries is to use bind parameters:

```
var query = "FOR doc IN @@someCollection" +
            " FILTER doc.@field == @userSuppliedValue" +
            " RETURN doc";
```

This separation of the query string from the bind parameters is not only safer, but makes query maintenance easier over time.

AQL also offers the retrieval of documents from arbitrary collections via the DOCUMENT function. The following query will extract two documents (with keys “one” and “two”) from the collections “test1” and “test2”.

```
FOR c IN ["test1", "test2"]
  FOR key IN ["one", "two"]
    RETURN DOCUMENT(c, key)
```

Bind parameters and other expressions can be used in the call to DOCUMENT, e.g.

```
FOR c IN [CONCAT("prod", @name), CONCAT("dev", @name)]
  FOR key IN @keys
    RETURN DOCUMENT(c, key)
```

However not all parts of a query can be made fully dynamic. For example, the following will not work:

```
FOR c IN ["prod", "dev"]
  FOR doc IN CONCAT(c, @name) // will not do a collection scan!
    RETURN doc
```

This is because the query optimizer that produces the query execution plan will select appropriate indexes. Additionally, query fragments may get distributed to different shards on different physical machines in a cluster setup.
All of that will be much harder to achieve if queries would be fully dynamic.

Q: How do we compare performance characteristics between AQL vs SQL? What can we do in AQL that we cannot in SQL and vice versa?

A: ArangoDB and AQL are (almost) schema-free while SQL and relational databases are “structured”.

For example, if the documents in collection “test” do not have any attribute “name”, then the following AQL query would still run, and produce “null” results for “doc.name”:

```
FOR doc IN test
  RETURN doc.name
```

The equivalent SQL query would fail.

This can be considered a disadvantage and an advantage at the same time. Some people dislike it, because typos in query strings may go unnoticed for a while. As shown above, it is not an error in ArangoDB to query a non-existing attribute. There may even be some documents in the collection that have that attribute and some that do not have the attribute.

However, this allows for flexible data structures, and also makes lazy upgrading of structures easier. For example, no ALTER TABLE or similar operation is required in ArangoDB to upgrade table/collection structures when a new attribute is added or old attributes are “outdated”.

In SQL you will always have a fixed table structure. If a flexible data layout is required, you will have to use the EAV (entity-attribute-value) pattern, which will quickly become slow and does not really support different data types, or you will have to store a custom format (e.g. JSON, XML, BLOBs) inside a column. This may work, but is a bit in contrast to the relational model.

Additionally, the schema-freedom in ArangoDB also allows putting together results quite flexible. For example, related documents can be returned as inline arrays:

```
FOR user IN users
  RETURN {
    user: user._key,
    watched: (
      FOR w IN watched   // subquery to fetch items watched
        FILTER w.userId == user._key
        RETURN w
    )
  }
```

```
[
  {
    "user" : "test1",
    "watched" : [
      {
        "_key" : "43666676",
        "episode" : 3,
        "season" : 2,
        "type" : "series",
```

```
        "user" : "test1"
      },
      {
        "_key" : "43666698",
        "name" : "Indespicable me",
        "released" : 2010,
        "type" : "movie",
        "user" : "test1"
      }
    ]
  }
]
```

ArangoDB and AQL also support built-in graph queries. This is useful in cases when connections with an unknown path depths have to be followed. In SQL, that can be done via joins, but the join depth is normally hard-coded in the query (number of join clauses used). It is hard in SQL to do arbitrary depth traversals into a graph without knowing the depth beforehand.

Q: I was reading ArangoDB manual and got a question about Sharding. In this book, it has a little information about Sharding. I was wondering if you can explain about shardkeys and indexkeys.

A: In a cluster setup, collections can be split into multiple shards. This is useful for collections that contain many documents. The number of shards can be specified when a collection is created, e.g.

```
db._create("test", {
  numberOfShards: 4
});
```

The sharding is done automatically, based on the specified shard key(s). If none are provided at collection creation, the default shard key attribute is "_key".

To shard by a different attribute (or combination of attributes), the shard key(s) can be specified at collection creation:

```
db._create("test", {
  numberOfShards: 4,
  shardKeys: ["country"]
});
```

In a cluster setup queries will be most efficient if the query also filters on the shard key(s).
Extra non-unique indexes can be created on other attributes than the shard keys.

Q: Elaborate on decision to not include SQL as another model. HA strategy/support AWS availability - native, managed service

A: There are multiple reasons why SQL is not another model in ArangoDB. First of all, SQL is powerful but very complex to implement and get right. So it would have been a tremendous effort to create a full-featured SQL interface for ArangoDB.
Then the structured of SQL is not a 100% match for the schema-free design of ArangoDB. ArangoDB is more flexible than SQL would allow, for example, when returning result sets.
The graph queries in ArangoDB would not be natively supported in SQL, so we would have to build custom extensions to SQL that would not be portable. Different relational database vendors have gone this way. They have put some graph functionality on top of their SQL implementations, but these are proprietary add-ons and mutually incompatible. So they do support a standard (SQL) but in fact there are subtle difference between the different SQL implementations.
We did not want to go that way but have a query language that allows for flexible and arbitrarily complex queries and that we can still extend if required.

HA strategy/support and AWS availability need to be covered elsewhere.
Just a quick note: there is an AMI for ArangoDB on AWS available at https://aws.amazon.com/marketplace/pp/B00RNJ092K

Q: I have a basic question, are there any good ways to upload data from rational db、txt 、csv to arangodb,

A: A good way to import data from CSV, TSV or JSON files into ArangoDB is to use arangoimp. This is an executable that is shipped with ArangoDB and can be run from the command line.

The invocation is as simple as:

```
arangoimp --type <type>
          --file <input file>
          --collection <collection name>
          --create-collection <true|false>
          --server.endpoint <endpoint>
          --server.username <username>
          --server.database <database name>
```

For example

```
arangoimp --type csv            --file data.csv
          --collection users    --create-collection true
          --server.endpoint     tcp://127.0.0.1:8529
          --server.username     root
```

The manual for arangoimp can be found here:
https://docs.arangodb.com/3.1/Manual/Administration/Arangoimp.html