NoSQL Data Models

Reading

- NoSQL Distilled[1] chapters 2, 3

Reading questions

- What does the term **data model** refer to?
- What are the four main **data models** in the NoSQL ecosystem?
- What does the term **aggregate** refer to? (section 2.1)
- How do we decide if we should use an aggregate-oriented data model vs an aggregate-ignorant one like relational databases? Where does each model excel?
- How do key-value stores and document databases differ in their approach to aggregate structures?
- How are column-family databases organized?
- What is schemaless design? What are its advantages and drawbacks?
- What are *materialized views*? What are their uses?

Notes

Aggregate Data Models

An **aggregate** is a collection of related objects that we treat as a unit. These aggregates tend to transcend the normalized breakdown that relational databases promote. For example, recall our basic evaluations database structure:

There are many different "aggregates" we could consider. Ultimately what determines an aggregate has to do with the kinds of queries one might want to perform.

- We could consider an "evaluation", which would contain in it the various questions and answers pertinent to that evaluation. In JSON terms, it may look something like this:

```
{
  "evalHash": ".....",
  "sectionId": ".....",
  "answers": [
    {
      "questionId": ".....",
      "questionPrompt": ".....",
      "questionType": ".....",
      "instructor": ".....",
      "answerNum": ".....",
      "answerText": "....."
```

[1] http://learning.acm.org/books/book_detail.cfm?id=2381014&type=safari

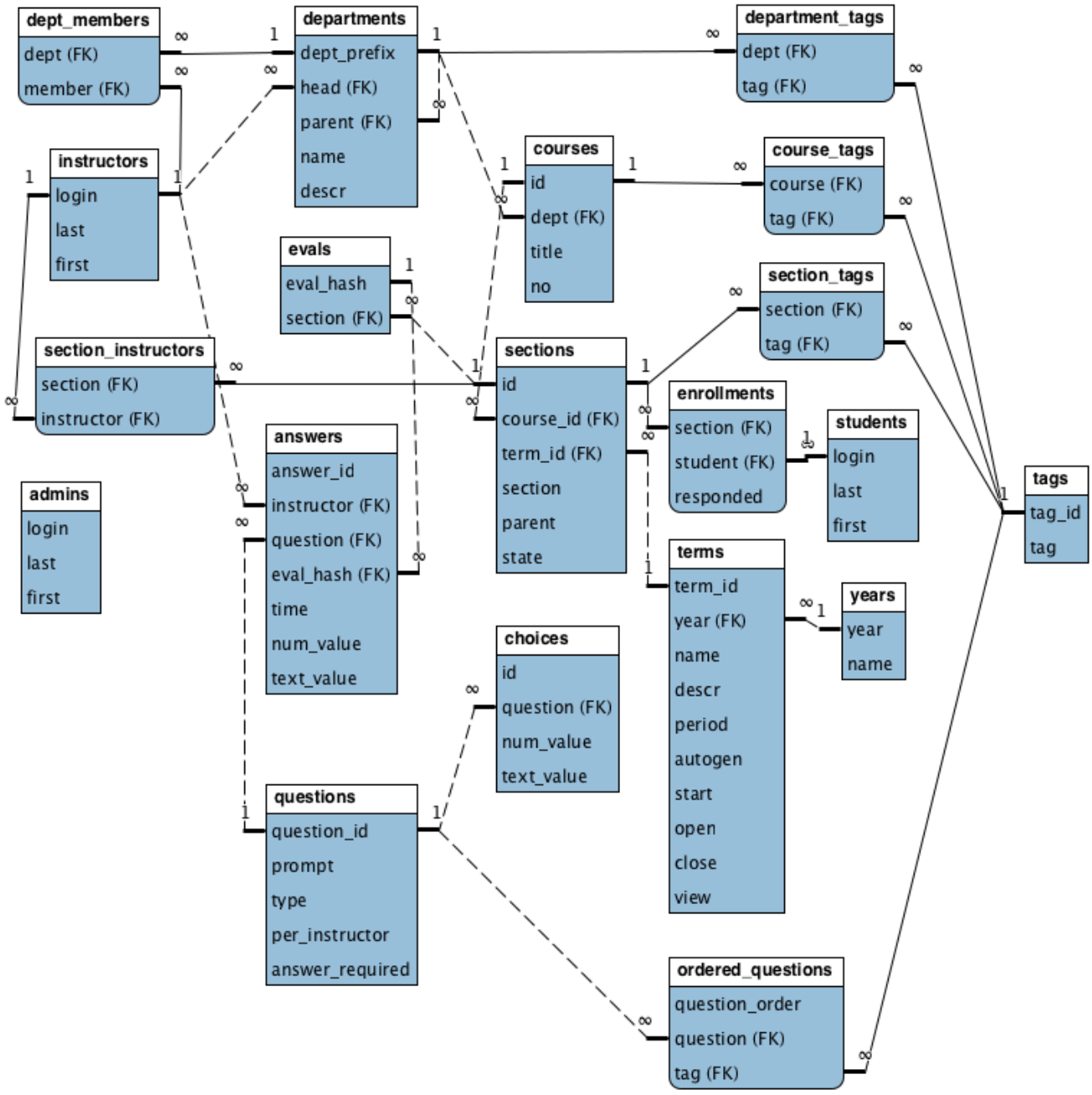

Figure 1: Evaluations Database Relationships

```
    },
    .....
  ]
}
```

- We could consider a "section" object, which would contain more details about the course and instructors, rather than simply linking to those tables. And it may further contain the various student enrollment lists and evaluation lists. So it may look something like this:

```
{
  "sectionId": ".....",
  "courseId": ".....",
  "fullName": ".....",
  "term": {
    "year": ".....",
    "period": "....."
  },
  "status": ".....",
  "instructors": [
    {
      "name": ".....",
      "id": "....."
    },
    .....
  ],
  "enrolled": [
    {
      "studentId": ".....",
      "studentName": ".....",
      "completed": "....."
    },
    .....
  ],
  "evaluations": [
    {
      "evalHash": "...",
    },
    {
      "evalHash": "...",
    },
    .....
  ]
}
```

The idea is that aggregates tend to put together all the information that will tend to be processed together. This simplifies and speeds up those queries, that respect this **unit of interaction**.

On the other hand, it also makes other queries more complicated. Suppose for example that we wanted to get summary information about all student answers to a particular question. With our aggregate example above, we have essentially two approaches:

- We could go through each "evaluation aggregate", extract the necessary information, and return it.
- If this is a frequent query, we could have created a different "answer" aggregate, that organizes the information in some alternate way. Then we would need to ensure somehow that the two structures remain synchronized, as we now store the same information in multiple places.

So bottom line: In *aggregate data models, some queries that respect the aggregate structure are fast, while other queries that transcend the structure become harder to perform.*

Key-Value Stores and Document Databases as Aggregate Data Models

Both key-value stores and document databases emphasize an aggregate-oriented design: Each aggregate is identified by its id/key and treated as a unit.

The difference between the two is on how the contents of the aggregate are treated:

- In a key-value store the contents/value of the aggregate are completely opaque: You can only retrieve an aggregate based on its key, you cannot look into the data and make sense of it. For many kinds of data this is the right thing.
- In a document database, we can submit queries based on the fields in the aggregate structure, not just solely on the key. The contents of the document/aggregate are therefore readable.