HBase Schema Design

HBase Schema Design

How I Learned To Stop Worrying And Love Denormalization

What Is **Schema Design**?

Who am I?

Ian Varley
Software engineer at Salesforce.com
@thefutureian

What Is **Schema Design**?

Logical Data Modeling

What Is **Schema Design**?

Logical Data Modeling

+

Physical Implementation

You always start with a logical model.
Even if it's just an implicit one.

That's totally fine. (If you're right.)

There are lots of ways to model data.
The most common one is:

Entity / Attribute / Relationship

(This is probably what you know
of as just "data modeling".)

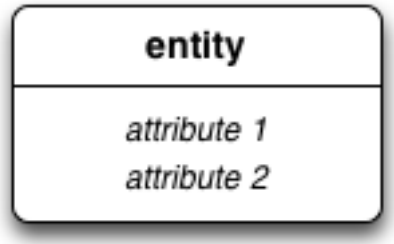
There's a well established visual language for data modeling.

Entities are boxes.

With rounded corners if you're fancy.

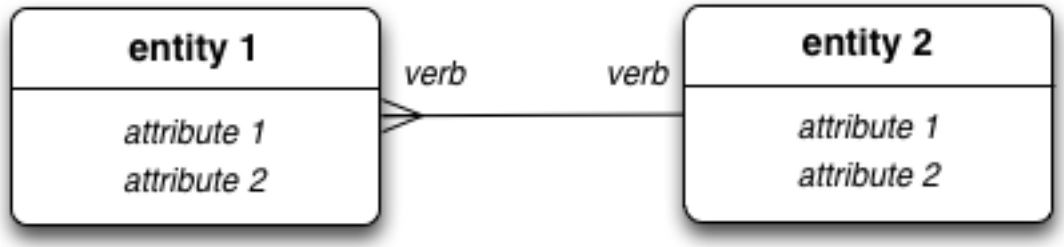
Attributes are listed vertically in the box.

Optionally with data types, for clarity.



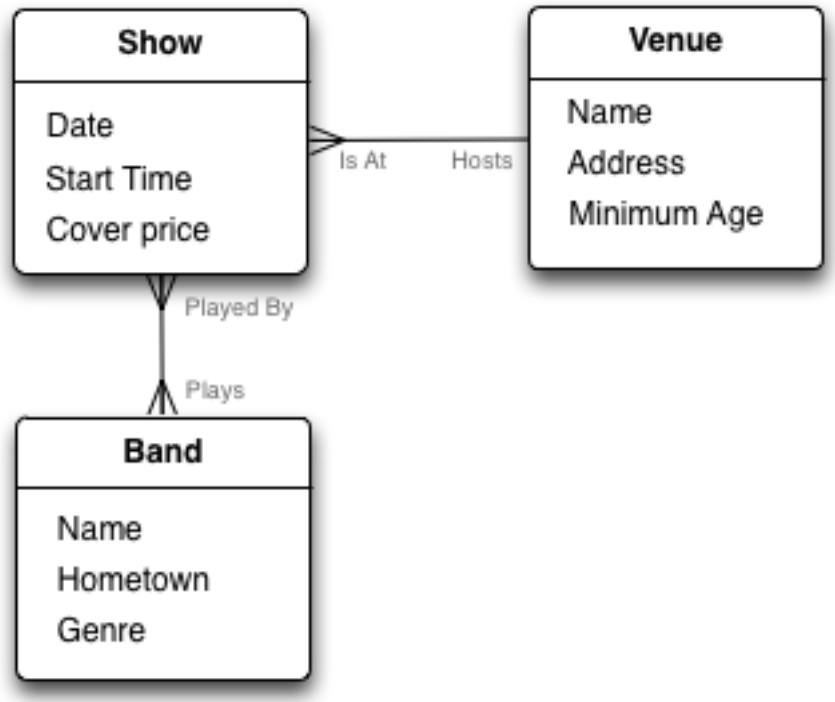
Relationships are connecting lines.

Optionally with special endpoints, and/or verbs

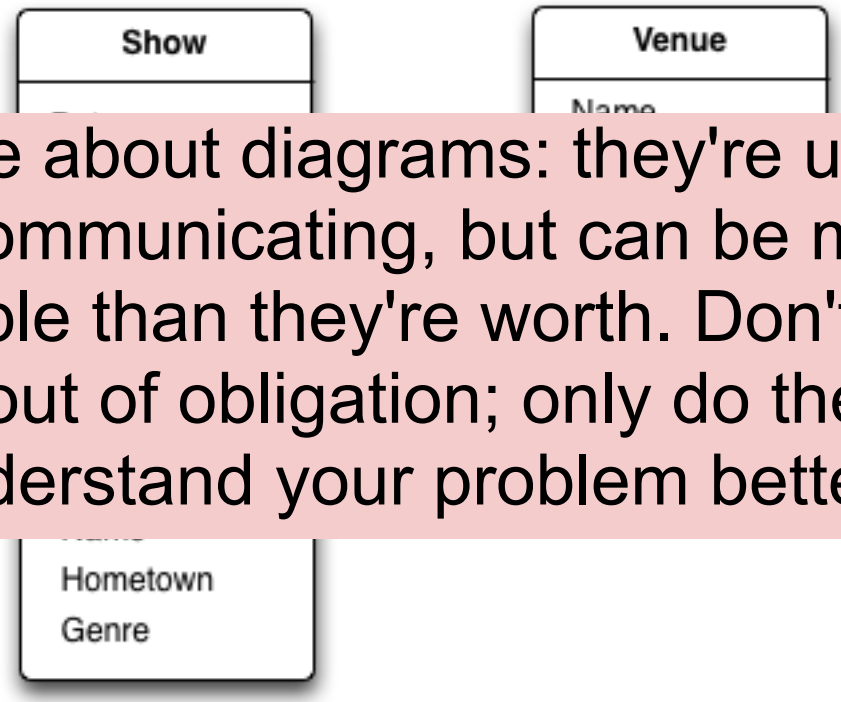


Example: **Concerts**

Example: **Concerts**

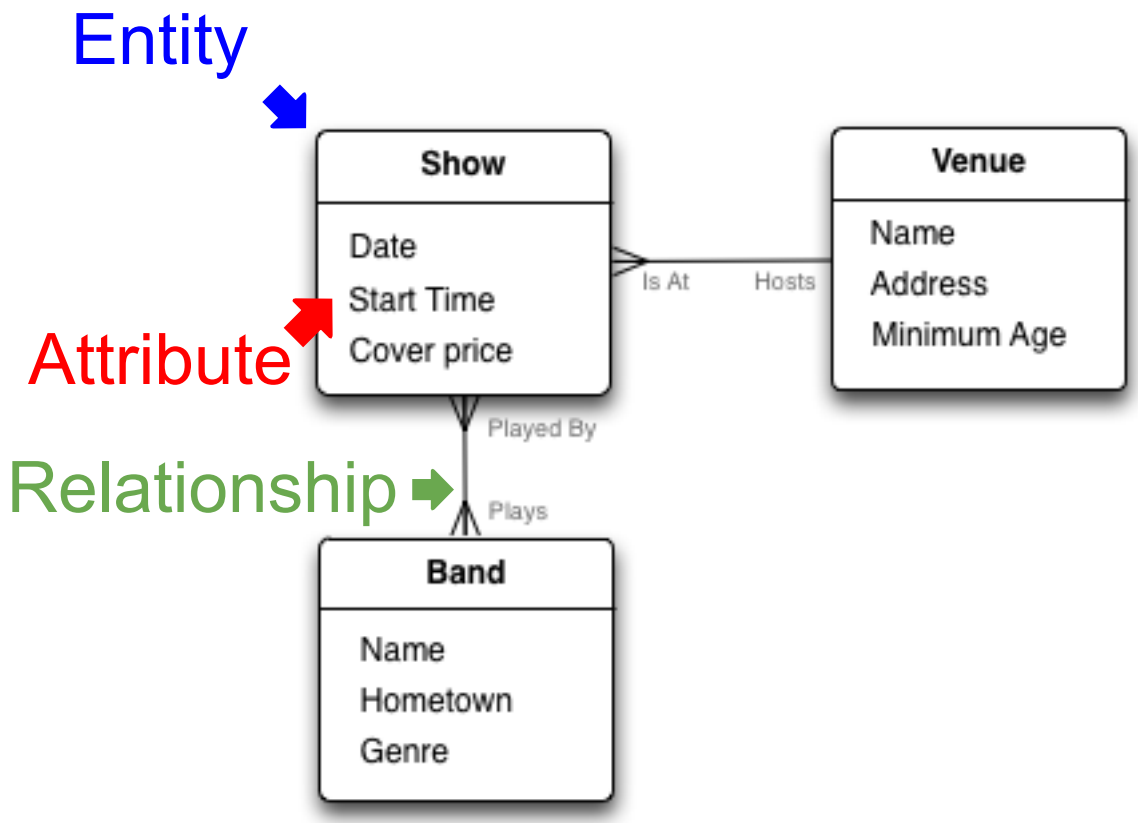


Example: **Concerts**



A note about diagrams: they're useful for communicating, but can be more trouble than they're worth. Don't do them out of obligation; only do them to understand your problem better.

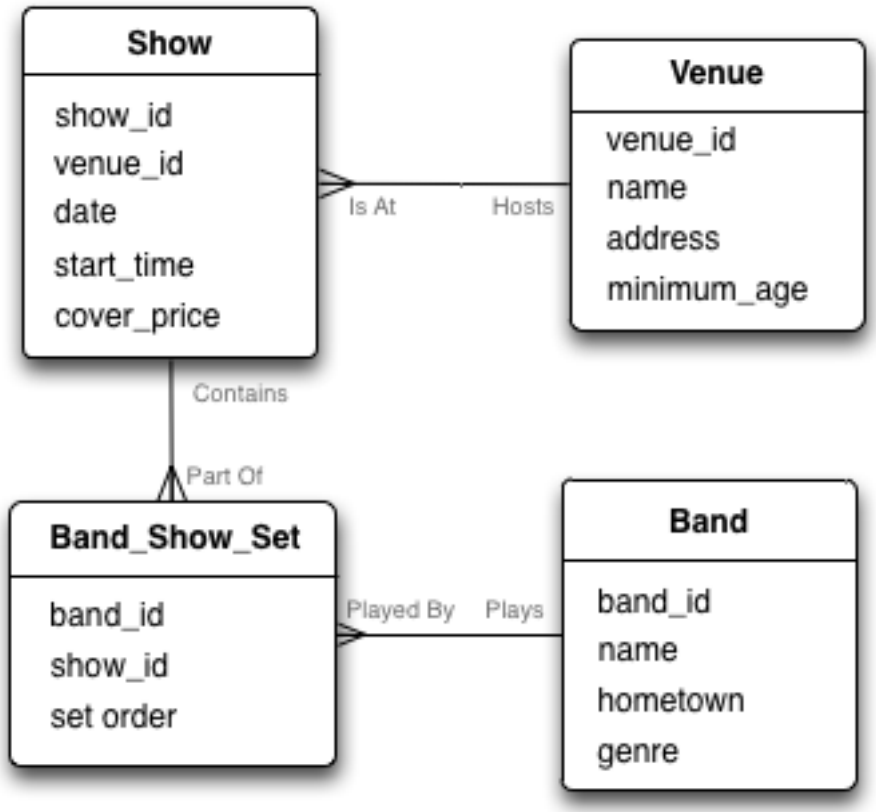
Example: **Concerts**



For relational databases, you usually start with this **normalized** model, then plug & chug.

For relational databases, you usually start with this **normalized** model, then plug & chug.

Entities → Tables
Attributes → Columns
Relationships → Foreign Keys
Many-to-many → Junction tables
Natural keys → Artificial IDs


Show
show_id
venue_id
date
start_time
cover_price
Is At
Hosts
Venue
venue_id
name
address
minimum_age
Contains
Part Of
Band_Show_Set
band_id
show_id
set order
Played By
Plays
Band
band_id
name
hometown
genre

So, what's missing?

So, what's missing?

If your data is not massive,

NOTHING.

You should use a relational database. They rock*

So, what's missing?

If your data is not massive,

NOTHING.

You should use a relational database. They rock*

* - This statement has not been approved by the HBase product management committee, and neglects known deficiencies with the relational model such as poor modeling of hierarchies and graphs, overly rigid attribute structure enforcement, neglect of the time dimension, and physical optimization concerns leaking into the conceptual abstraction.

Relational DBs work well because they are close to the pure logical model.

That model is less likely to change as your business needs change. You may want to ask different questions over time, but if you got the logical model correct, you'll have the answers.

Ah, but what if you **do** have massive data? Then what's missing?

Problem: The relational model doesn't acknowledge scale.

Problem: The relational model doesn't acknowledge scale.

"It's an implementation concern;
you shouldn't have to worry about it."

The trouble is, you **do** have to worry about it. So you...

- Add indexes
- Add hints
- Write really complex, messy SQL
- Memorize books by Tom Kyte & Joe Celko
- Bow down to the optimizer!
- Denormalize
- Cache
- etc ...

Generally speaking, you **poke holes** in the abstraction, and it starts leaking.

So then you hear about this thing called **NoSQL**. Can it help?

Maybe. But ultimately, it's just a different way of physically representing your same logical data model.

Some things are easier; some are much harder.

If you haven't started by understanding your logical model, you're doing it wrong.

HBase Architecture: A Brief Recap

HBase Architecture: A Brief Recap

- Scales by splitting all rows into regions

HBase Architecture: A Brief Recap

- Scales by splitting all rows into regions
- Each region is hosted by exactly one server

HBase Architecture: A Brief Recap

- Scales by splitting all rows into regions
- Each region is hosted by exactly one server
- Writes are held (sorted) in memory until flush

HBase Architecture: A Brief Recap

- Scales by splitting all rows into regions
- Each region is hosted by exactly one server
- Writes are held (sorted) in memory until flush
- Reads merge rows in memory with flushed files

HBase Architecture: A Brief Recap

- Scales by splitting all rows into regions
- Each region is hosted by exactly one server
- Writes are held (sorted) in memory until flush
- Reads merge rows in memory with flushed files
- Reads & writes to a single row are consistent

So what does data in HBase look like?

HBase Data Model: Brief Recap

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic byte array, with one row key

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic byte array, with one row key

Not just a bunch of bytes, dude! A k/v map!

HBase Data Model: Brief Recap

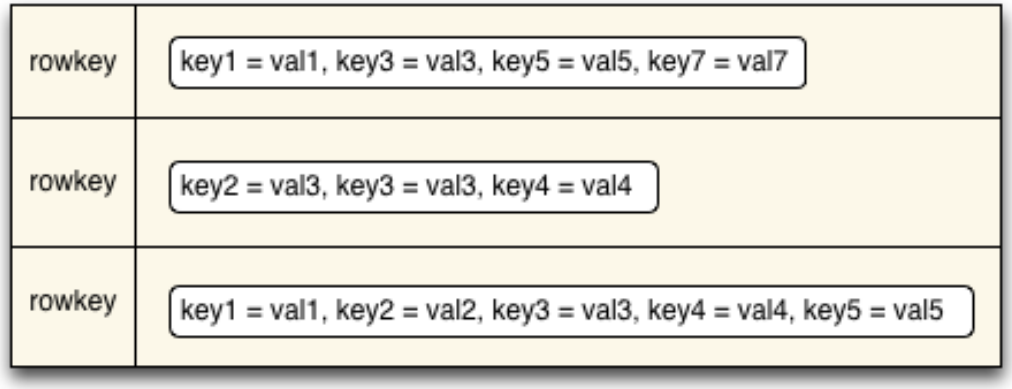
Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Value: a value in the k/v container inside a row



HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Value: a value in the k/v container inside a row

HBase Data Model: Brief Recap

Hold up! What about time?

...ime namespace, has many rows.
...ey/value container, with one row key
Column: a key in the k/v container inside a row
Value: a value in the k/v container inside a row

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

HBa ecap

Table: y rows.
Row: at row key
Colum a row
Timesta scending
Value: a ontainer

The "row" is atomic, and gets flushed to disk periodically. But it doesn't have to be flushed into just a single file!

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

It can be broken up into different store files with different properties, and reads can look at just a subset.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

This is called "Column Families". It's kind of an advanced design option, so don't think too much about it yet.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

From the Percolator paper by Google: "Bigtable allows users to control the performance characteristics of the table by grouping a set of columns into a locality group." That's a good way to think about CFs.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

Calling these "columns" is an unfortunate use of terminology. They're not fixed; each row can have different keys, and the **names are not defined at runtime**. So you can represent another axis of data (in the **key** of the key/value pair). More on that later.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

They're officially called "column qualifiers". But many people just say "columns".

Or "CQ". Or "Quallie". Now you're one of the cool kids.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

What data types are stored in key/value pairs?

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

What data types are stored in key/value pairs?

It's all **bytes**.

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

What data types are stored in key/value pairs?

Row keys, column names, values: arbitrary bytes

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

What data types are stored in key/value pairs?

Row keys, column names, values: arbitrary bytes
Table and column family names: printable characters

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

What data types are stored in key/value pairs?

Row keys, column names, values: arbitrary bytes
Table and column family names: printable characters
Timestamps: long integers

HBase Data Model: Brief Recap

Table: design-time namespace, has many rows.
Row: atomic key/value container, with one row key
Column Family: divide columns into physical files
Column: a key in the k/v container inside a row
Timestamp: long milliseconds, sorted descending
Value: a time-versioned value in the k/v container

HBase Data Model: Brief Recap

One more thing that bears repeating: every "cell" (i.e. the time-versioned value of one column in one row) is stored "fully qualified" (with its full rowkey, column family, column name, etc.) on disk.

So now you know what's available.
Now, how do you model things?

Let's start with the entity / attribute / relationship modeling paradigm,

and see how far we get applying it to HBase.

A note about my example: it's for clarity, not realism.

For bands & shows, there's not enough data to warrant using HBase, even if you're tracking every show by every band for all of human history. It might be GB, but not TB.

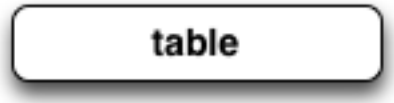
So, back to entities (boxes).

With fancy rounded corners.

What's that in HBase? A table, right?

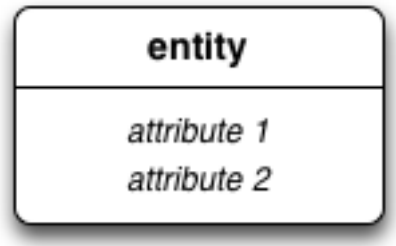
What's that in HBase? A table, right?

Dramatic foreshadowing: not always ...



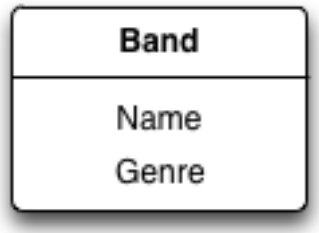
And what do entities have?

Attributes!



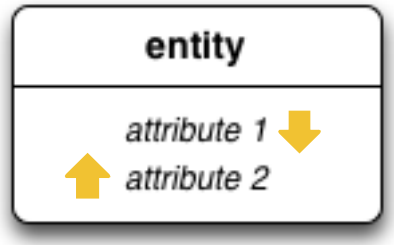
Attributes!

For example: a "Band" entity might have a "Name" (with values like "Jonathan Coulton") and a "Genre" (with values like "Nerd Core")



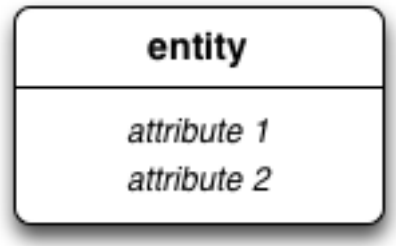
Logically, attributes are **unordered**.

Their vertical position is meaningless. They're simply **characteristics** of the entity.



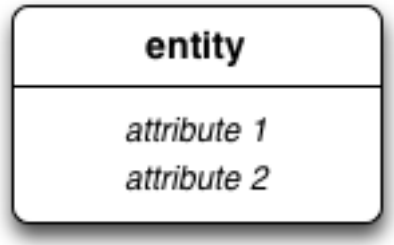
Attributes can be **identifying**.

i.e. they uniquely identify a particular instance of that entity.



Attributes can be **identifying**.

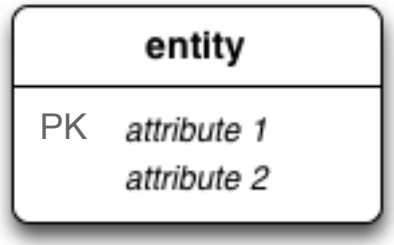
i.e. they uniquely identify a particular instance of that entity.



Logical models usually leave this out, and identity is implied (i.e. we just assume there's ***some*** set of attributes that's identifying, but we don't have to say what it is)

Attributes can be **identifying**.

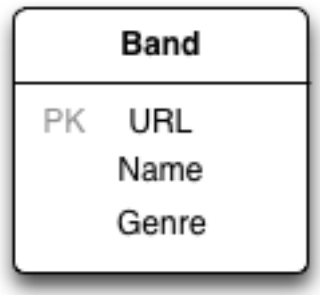
i.e. they uniquely identify a particular instance of that entity.



Physical models refer to it explicitly, like how in a relational database model you'd label some set of attributes as being the "Primary Key" (PK).

Attributes can be **identifying**.

i.e. they uniquely identify a particular instance of that entity.

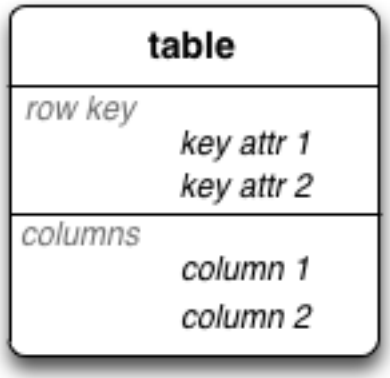


So in our Band example, neither Name nor Genre is uniquely identifying, but something like URL might be.

How does this map to HBase?

How does this map to HBase?

Identifying attributes (aka the "PK") become parts of the row key, and other attributes become columns.

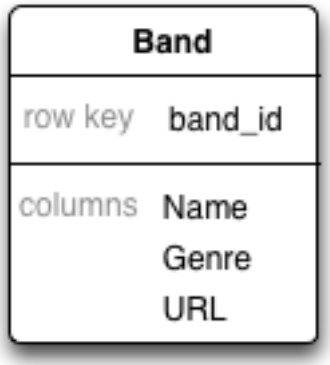


So here's a Band schema.

The row key is the URL, and we have Name and Genre as attributes.

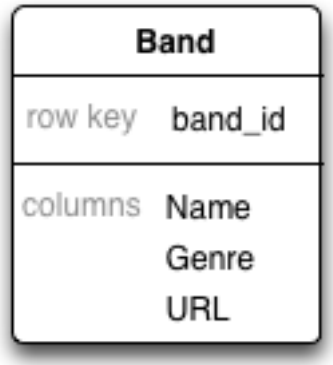
A much common pattern is to use IDs.

That way, you have an immutable way to refer to this entity forever, even if they leave MySpace.



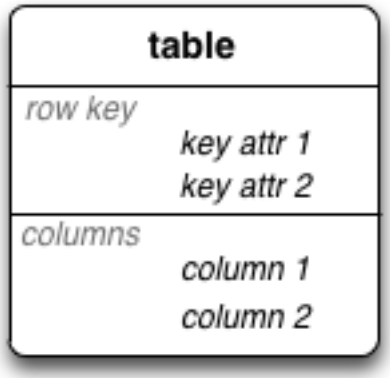
A much common pattern is to use IDs.

That way, you have an immutable way to refer to this entity forever, even if they leave MySpace.



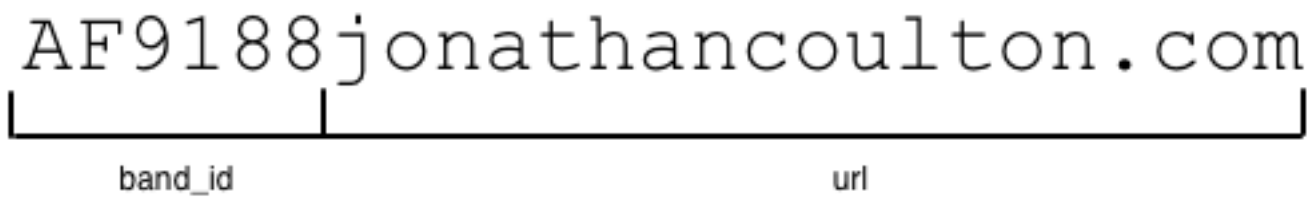
Where do IDs come from? We'll talk about that later.

If there's just a single row key,
how can there be multiple identifying attributes?



Let's call that "mashing".

Which is to say, concatenation, in either a fixed-width or delimited manner, in the byte array.



Mashing includes several techniques:

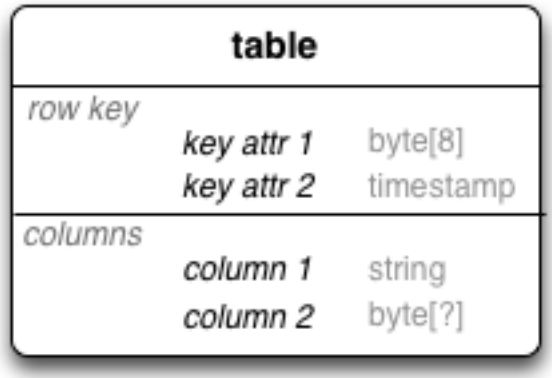
- Fixed byte width representation
- Delimiting (for variable length)
- Other serialization (avro, etc.)

Mashing includes several techniques:

- Fixed byte width representation
- Delimiting (for variable length)
- Other serialization (avro, etc.)

Doesn't matter how you do it; the important thing is that any byte array in HBase can represent more than one logical attribute.

If we want, we can even add types to the schema definition.



If we want, we can even add types to the schema definition.

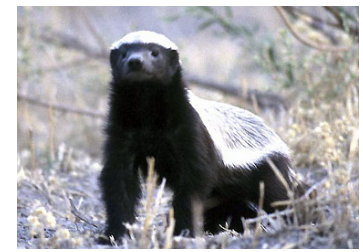
table		
row key		
	key attr 1	byte[8]
	key attr 2	timestamp
columns		
	column 1	string
	column 2	byte[?]

?

If we want, we can even add types to the schema definition.

table		
row key		
	key attr 1	byte[8]
	key attr 2	timestamp
columns		
	column 1	string
	column 2	byte[?]

?



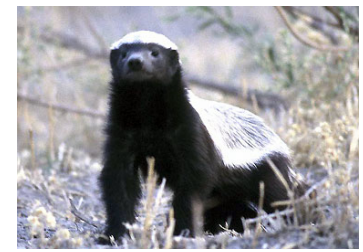
HBase don't care,
but we do (sometimes).

If we want, we can even add types.

You could also mark things as ASC or DESC, depending on whether you invert the bits.

table		
row key		
	key attr 1	byte[8]
	key attr 2	timestamp
columns		
	column 1	string
	column 2	byte[?]

?



This is pretty textbook stuff, but here's where it gets exciting.

This is pretty textbook stuff, but here's where it gets exciting.

(If you're astute, you'll notice we haven't talked about **relationships** yet.)

HBase has no foreign keys, or joins, or cross-table transactions.

This can make representing relationships between entities ... tricky.

Part of the beauty of the **relational** model is that you can punt on this question.

If you model the entities as fully normalized, then you can write any query you want at run time, and the DB performs **joins** on tables as optimally as it can.

(This relies **heavily** on indexing.)

In HBase (or any distributed DB)
you don't have that luxury.

Joins get way more expensive and complicated across a distributed cluster of machines, as do the indexes that make them happen efficiently.

HBase has neither joins nor indexes.

You have two choices, if you need relationships between entities.

You have two choices, if you need relationships between entities.

- Roll your own joins

You have two choices, if you need relationships between entities.

- Roll your own joins
- Denormalize the data

Rolling your own joins is hard.

You'll probably get it wrong
if you try to do it casually.

Ever implemented a multi-way merge sort by hand?

Denormalizing is fun!

But it also sucks. We'll see why.

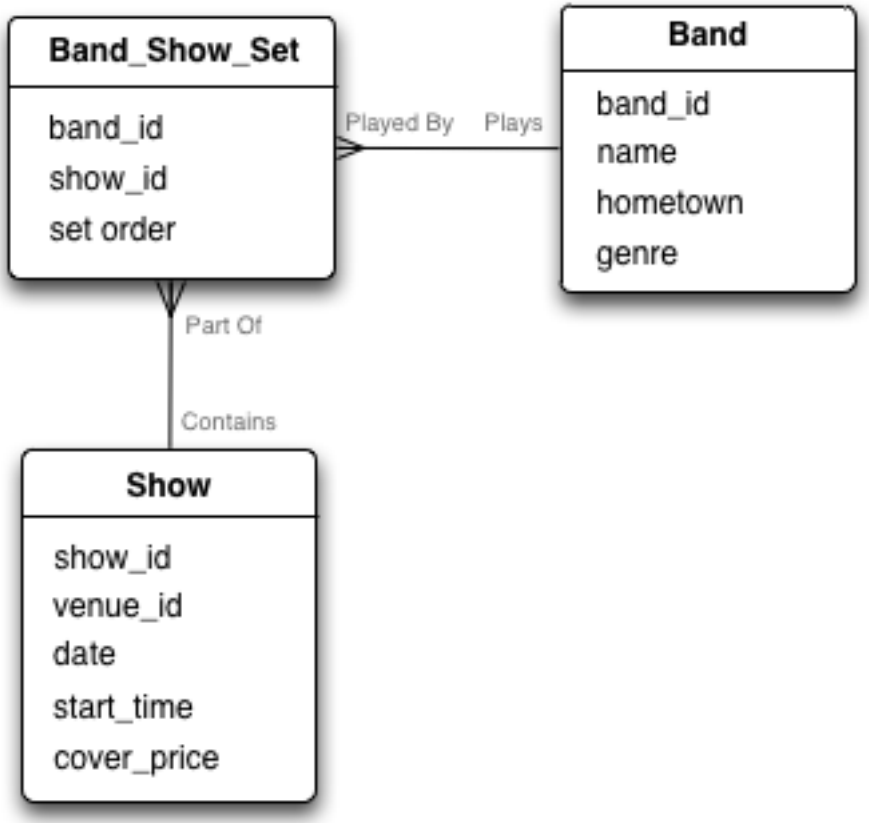
The basic concept of denormalization is simple:
two **logical** entities share
one **physical** representation.

The basic concept of denormalization is simple:
two **logical** entities share
one **physical** representation.

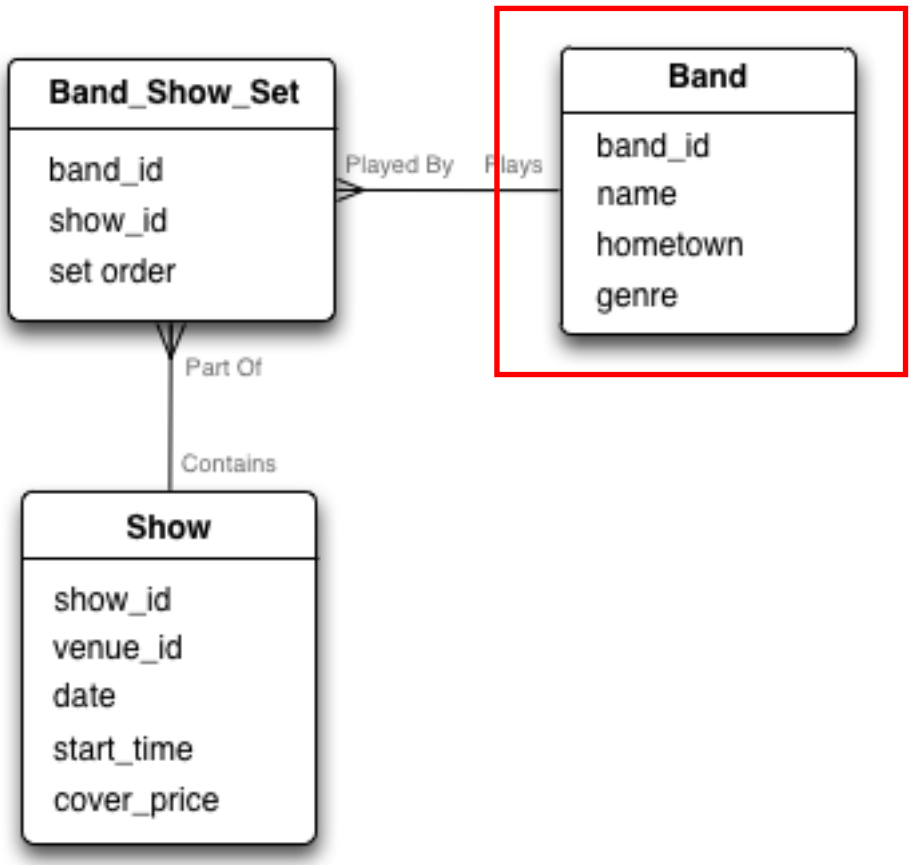
Of course, there are lots of ways to skin a cat...

Let's look at standard relational database denormalization techniques first.

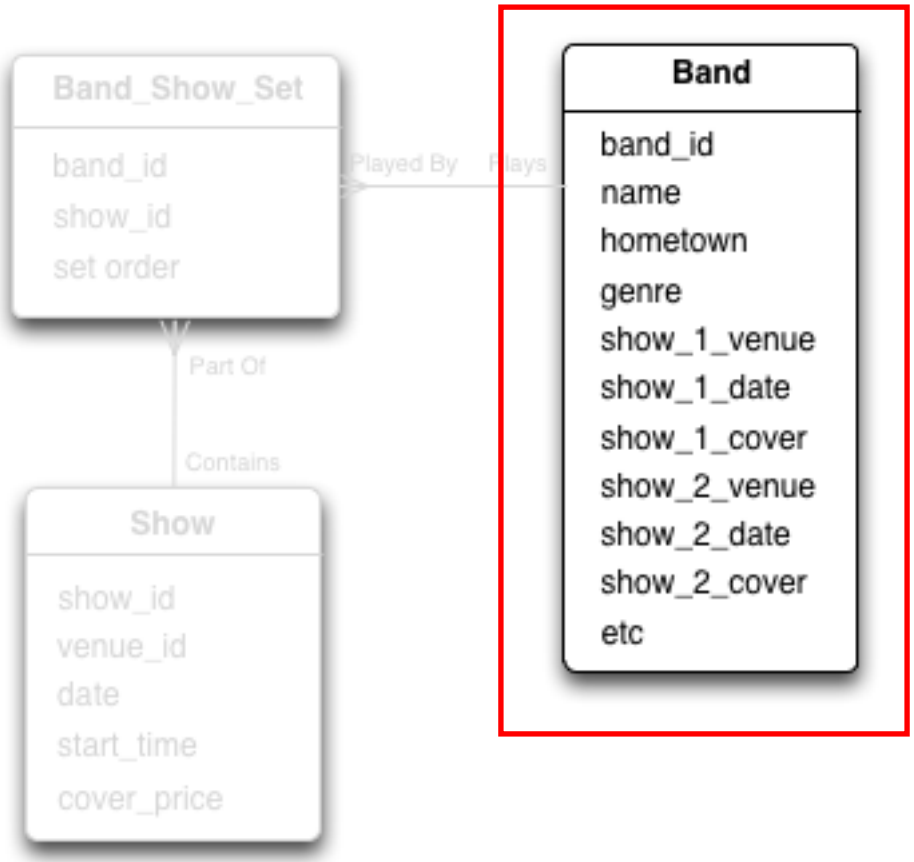
In a relational database, if the normalized physical model is this:



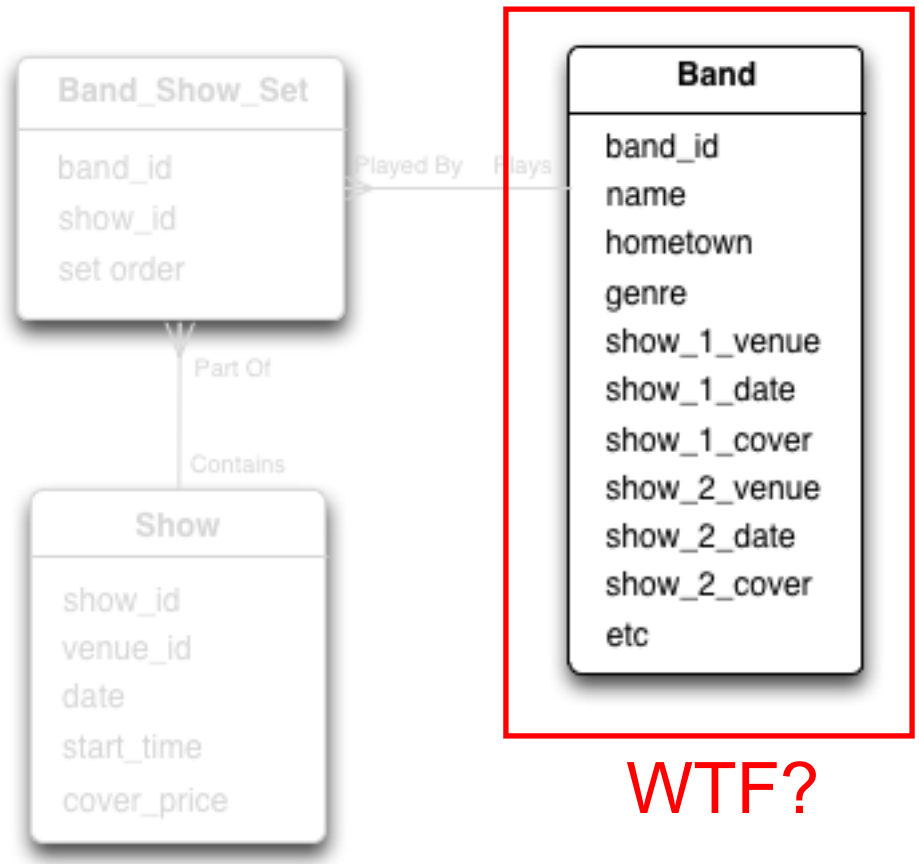
You could start with the Band, and give it extra columns to hold the shows:



You could start with the Band, and give it extra columns to hold the shows:

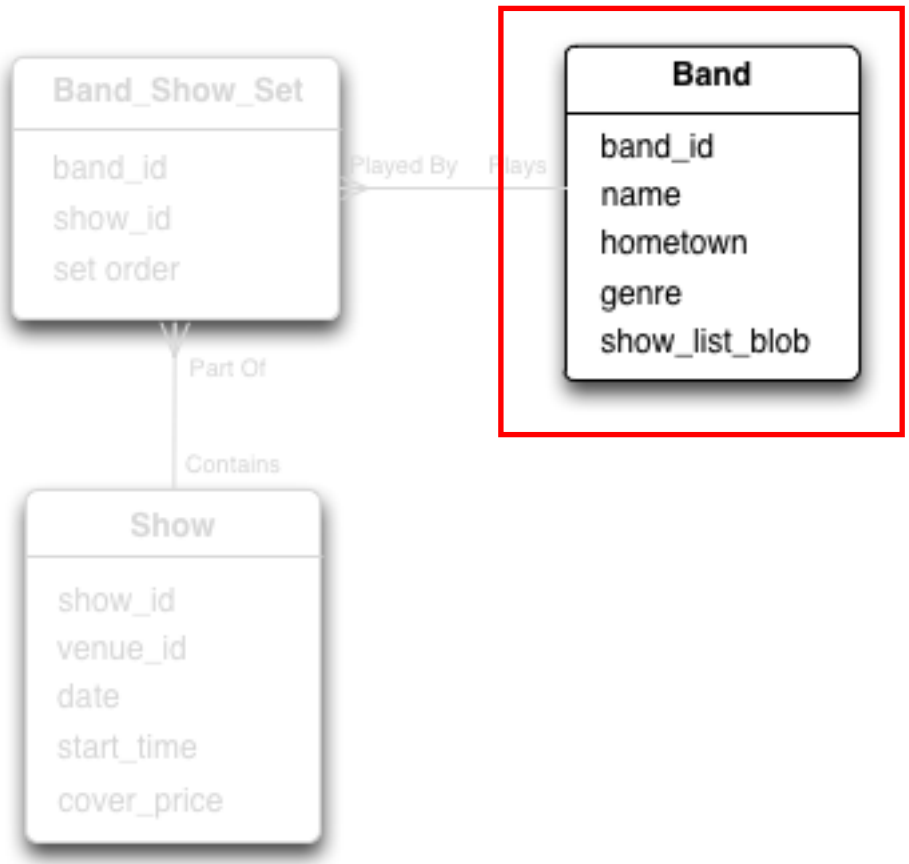


That's pretty obviously a bad idea, because bands can play a lot of shows.

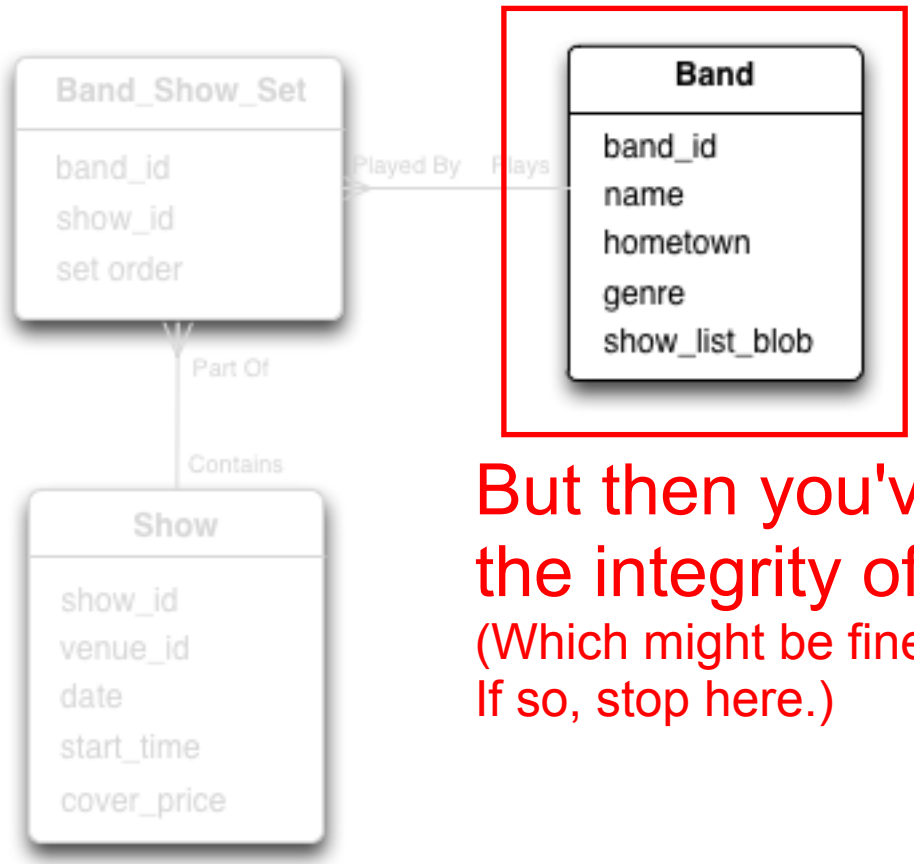


WTF?

You could also just give it an unstructured blob of text for all the shows.



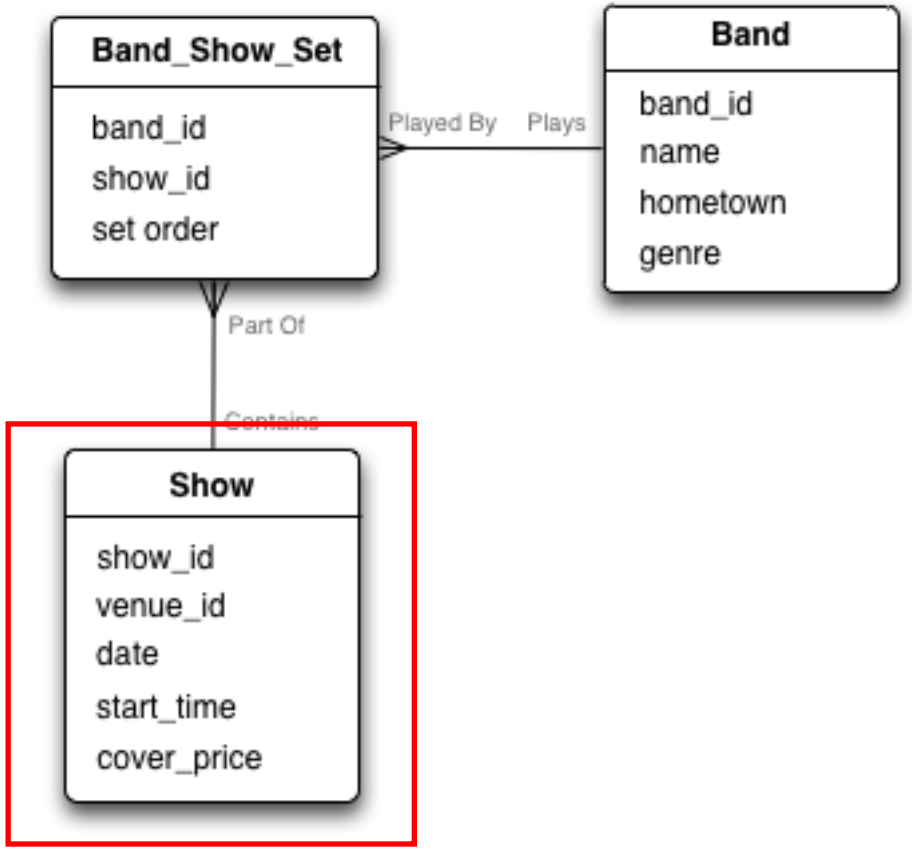
You could also just give it an unstructured blob of text for all the shows.



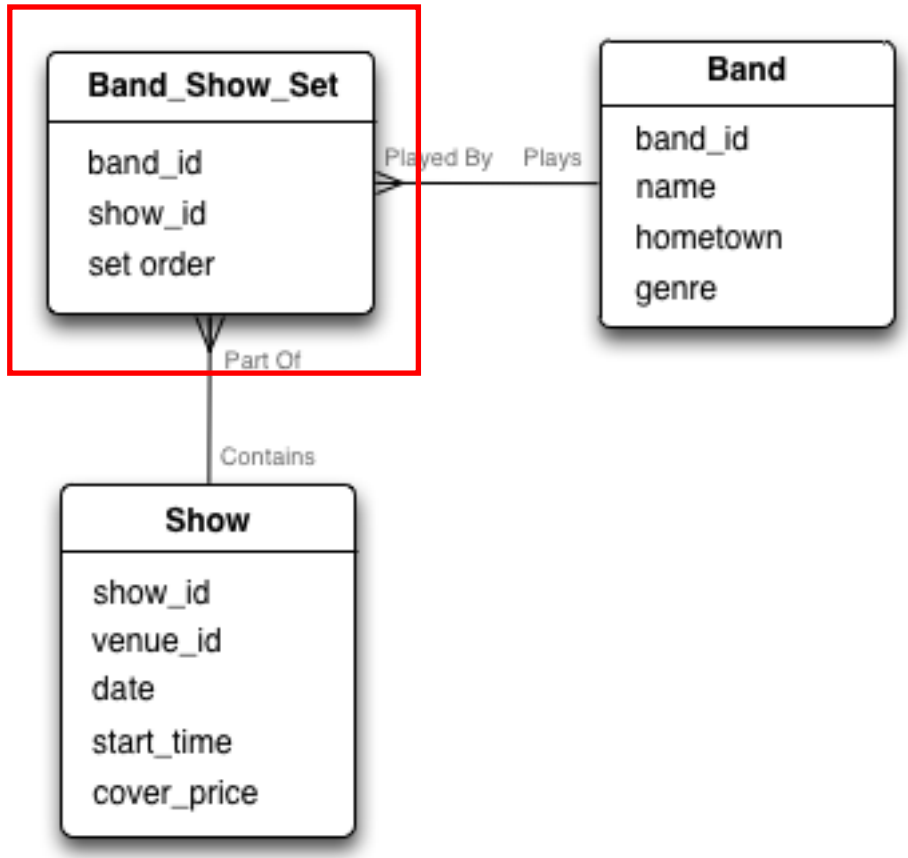
But then you've given up the integrity of your data.

(Which might be fine.
If so, stop here.)

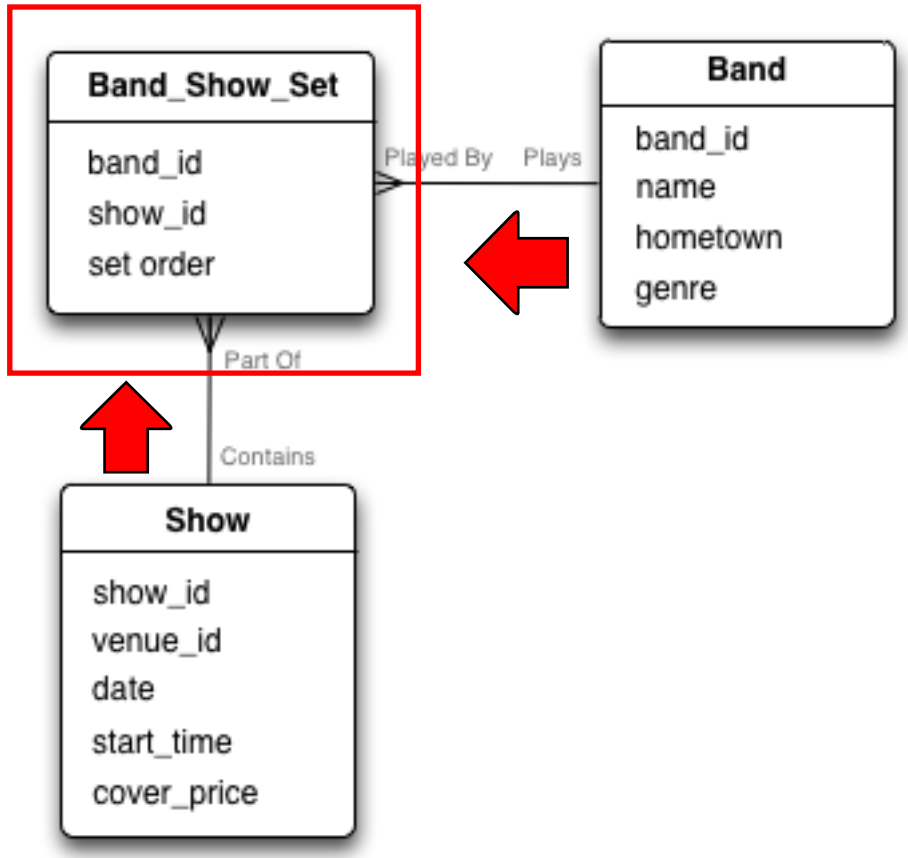
You get similar problems if you try to bring all the info into the Show table.



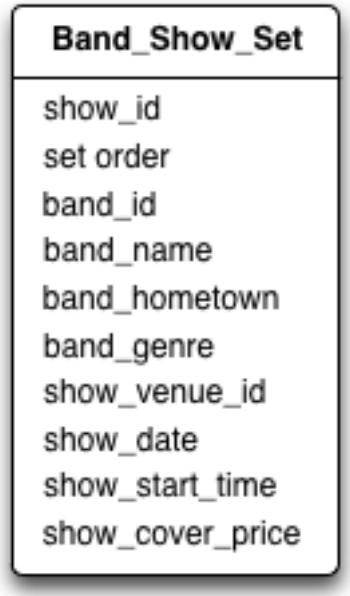
Another solution is to focus on the junction table.



And pull in copies of the info in the other tables:



Leaving you with one table, with one row per band/show combination:



The cons to this should be self-evident.

Updating and querying are more complicated, because you have to deal with many representations of the "same" data, which could get out of sync, etc.

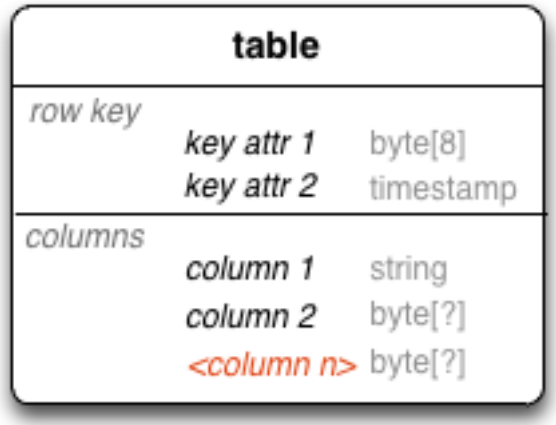
But the pros are that with **huge** data, answering some questions is much faster.

(If you happen to be asking the query in the same way you structured the denormalization, that is.)

So back to HBase. How do you denormalize in an HBase schema?

HBase columns can be defined at runtime.

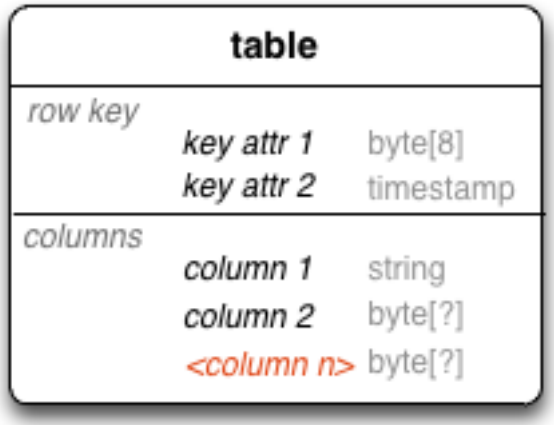
A column doesn't have to represent a pre-defined attribute.



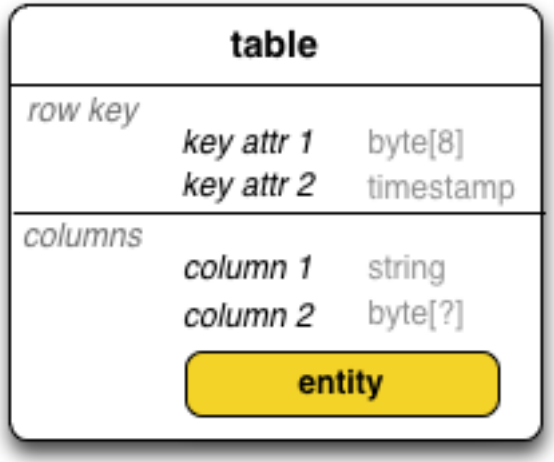
HBase columns can be defined at runtime.

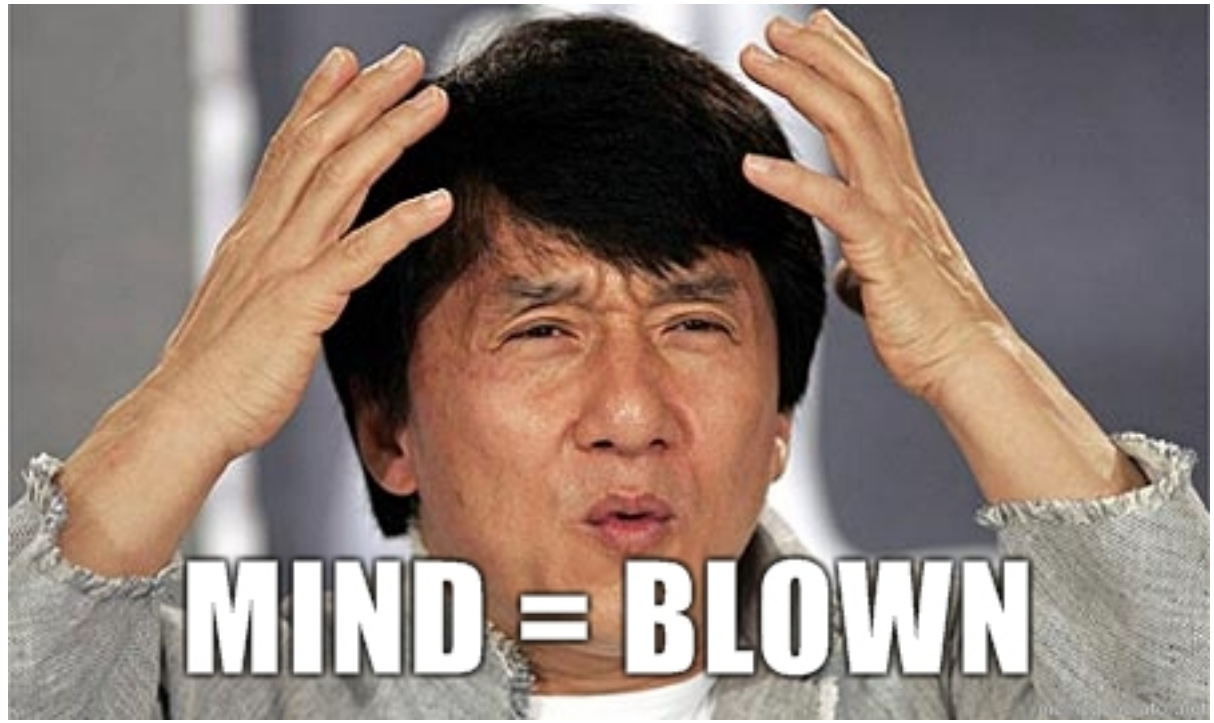
A column doesn't have to represent a pre-defined attribute.

In fact, it doesn't have to be an **attribute** at all.



A set of dynamically named columns can represent **another entity**!

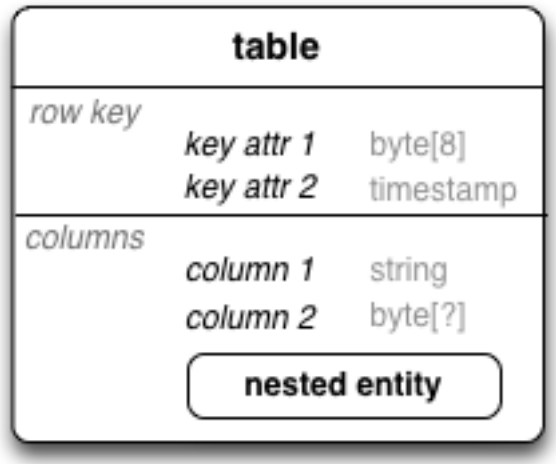



MIND = BLOWN


SOFTWARE

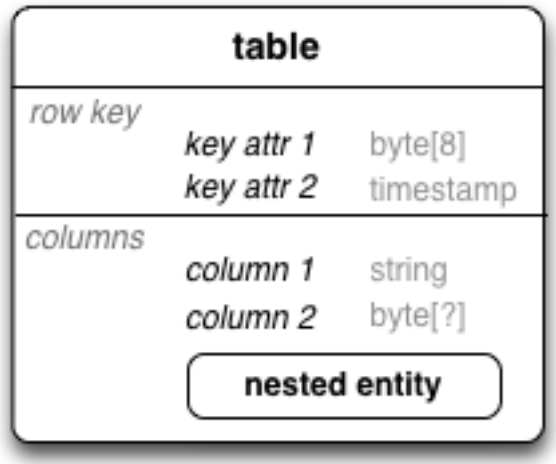
If you put data into the column name,

and expect many such columns in the same row,
then logically, you've created a **nested entity**.



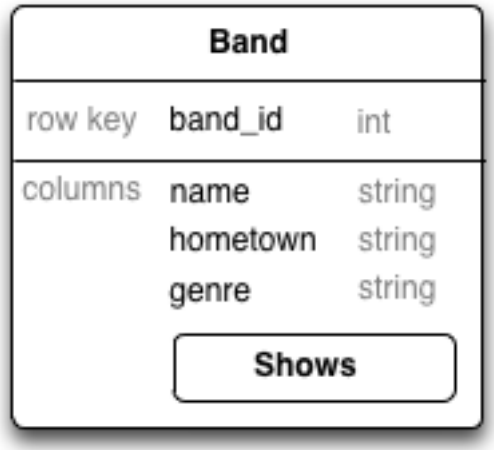
You can scan over columns.

See: hadoop-hbase.blogspot.com/2012/01/hbase-intra-row-scanning.html



So we can store shows inside bands.

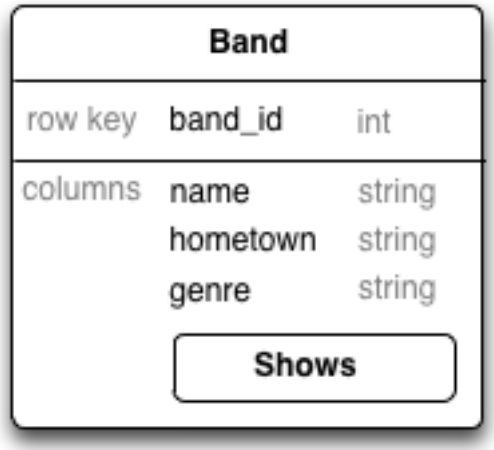
Which is like the denormalization we say earlier,
except without the relational DB kludges.



Note!

HBase don't care.

It's just a matter of how **your app** treats the columns. If you put repeating info in column names, you're doing this.



Why is this so difficult for most relational database devs to grok?

Because
relational databases have no concept of nested entities!

You'd make it a separate table in an RDBMS, which is more flexible but much more difficult to optimize.

It's very difficult to talk about HBase schemas if you don't acknowledge this.

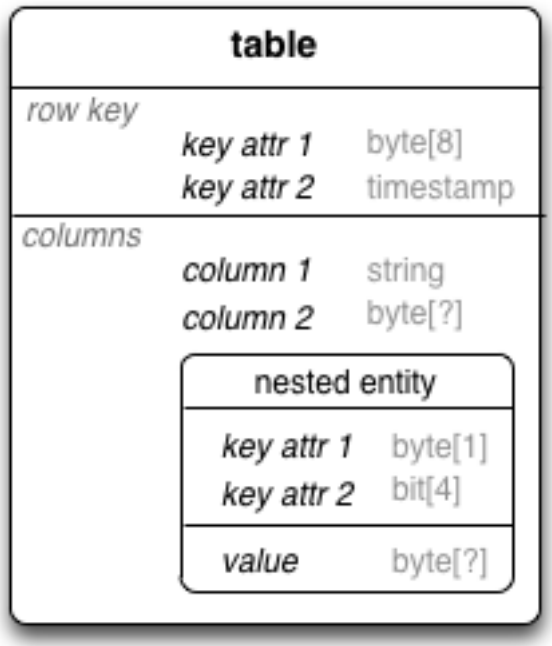
You don't have to represent it this way (with a nested box) but you have to at least acknowledge it. And maybe name it.

But, once you do acknowledge it,

you can do some neat things.

Nested entities can have attributes,

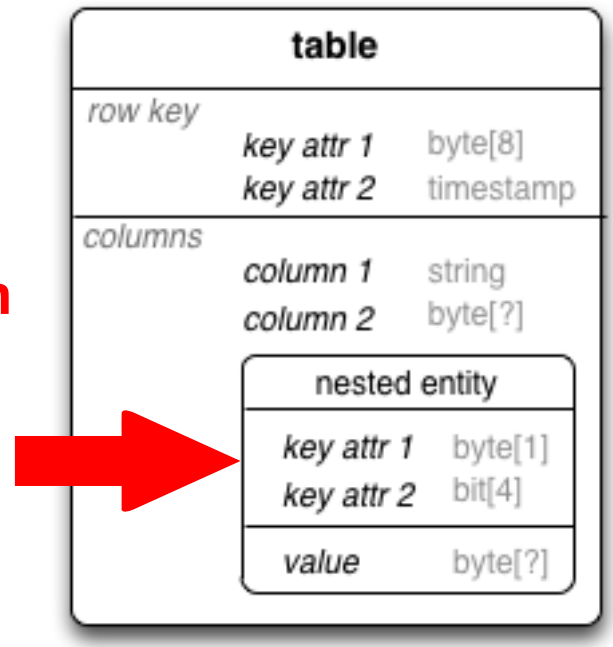
some of which are identifying.



Nested entities can have attributes,

some of which are identifying.

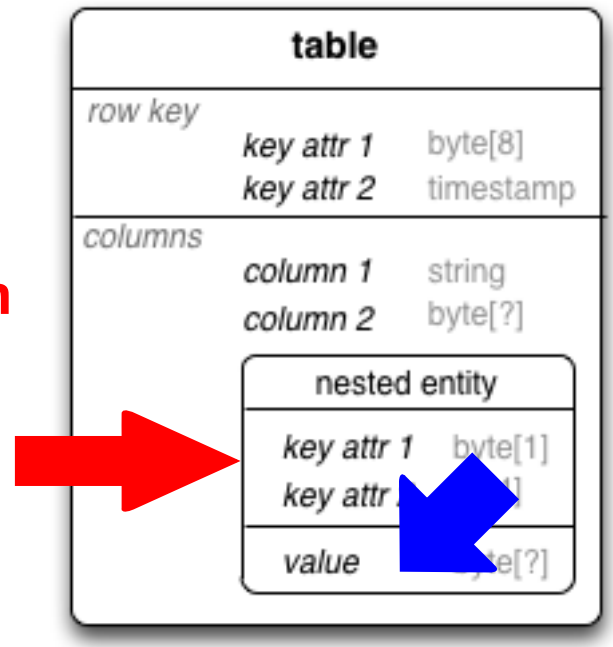
Identifying attributes make up the **column qualifier** (just like row keys, it could be multiple attributes mashed together)



Nested entities can have attributes,

some of which are identifying.

Identifying attributes make up the **column qualifier** (just like row keys, it could be multiple attributes mashed together)

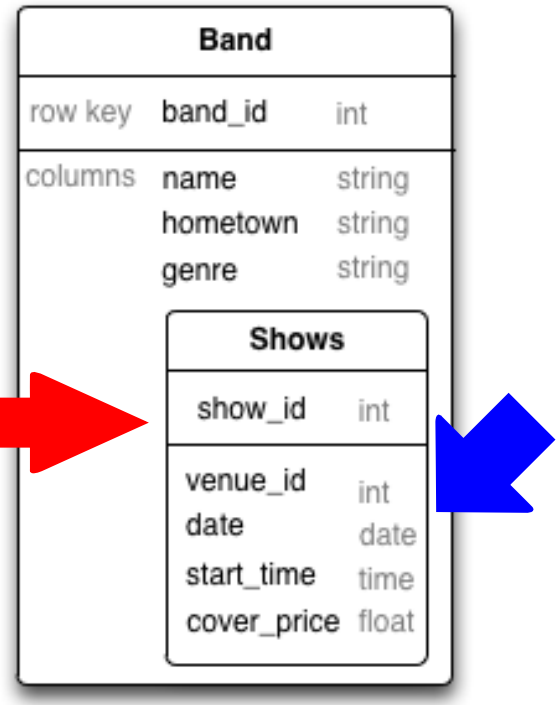


Non-identifying attributes are held in the value (again, you could mash many attributes in here)

Shows is nested in **Band**

show_id is the column qualifier

Other attributes are mashed into the value

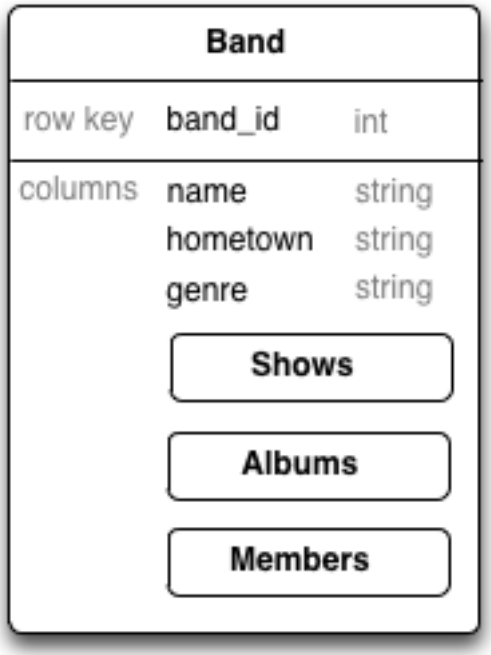


The column qualifier is the show id.

Everything else gets mashed into the value field.

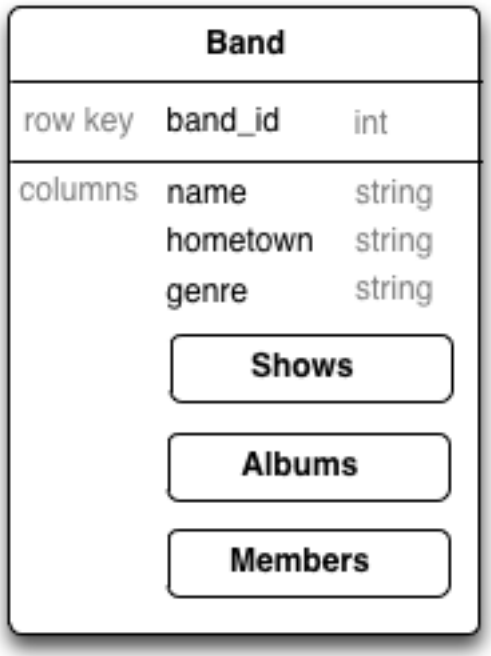
1 table can have **many** nested entities,

provided your app can tell them apart.



How do you tell them apart?

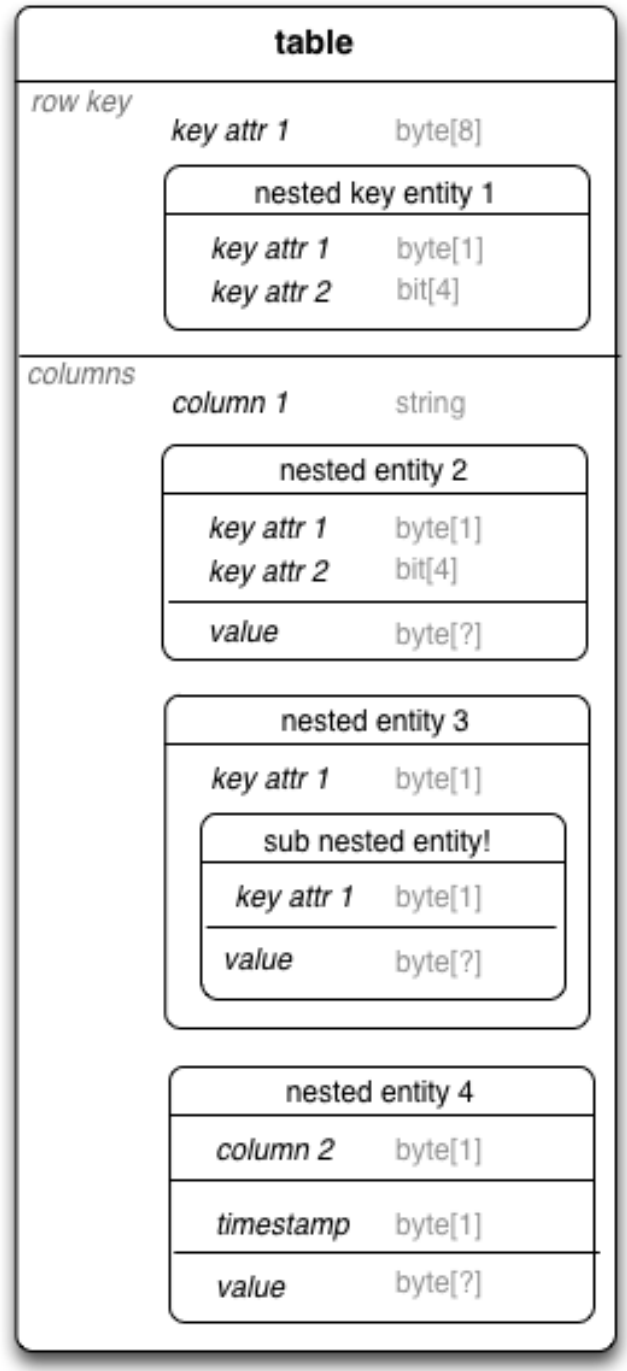
With prefixes ...



qualifier starts with:
"s" + show_id
"a" + album_id
"m" + name

Where can you nest entities?

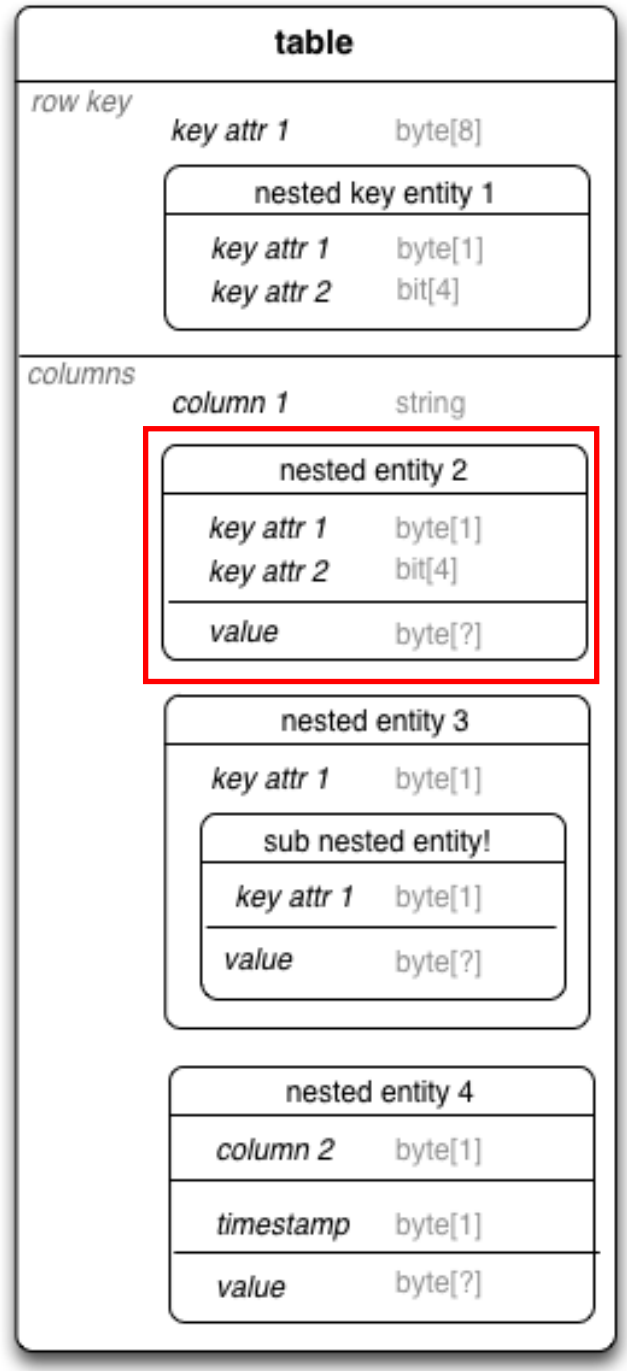
Knock yourself out.



Where can you nest entities?

Knock yourself out.

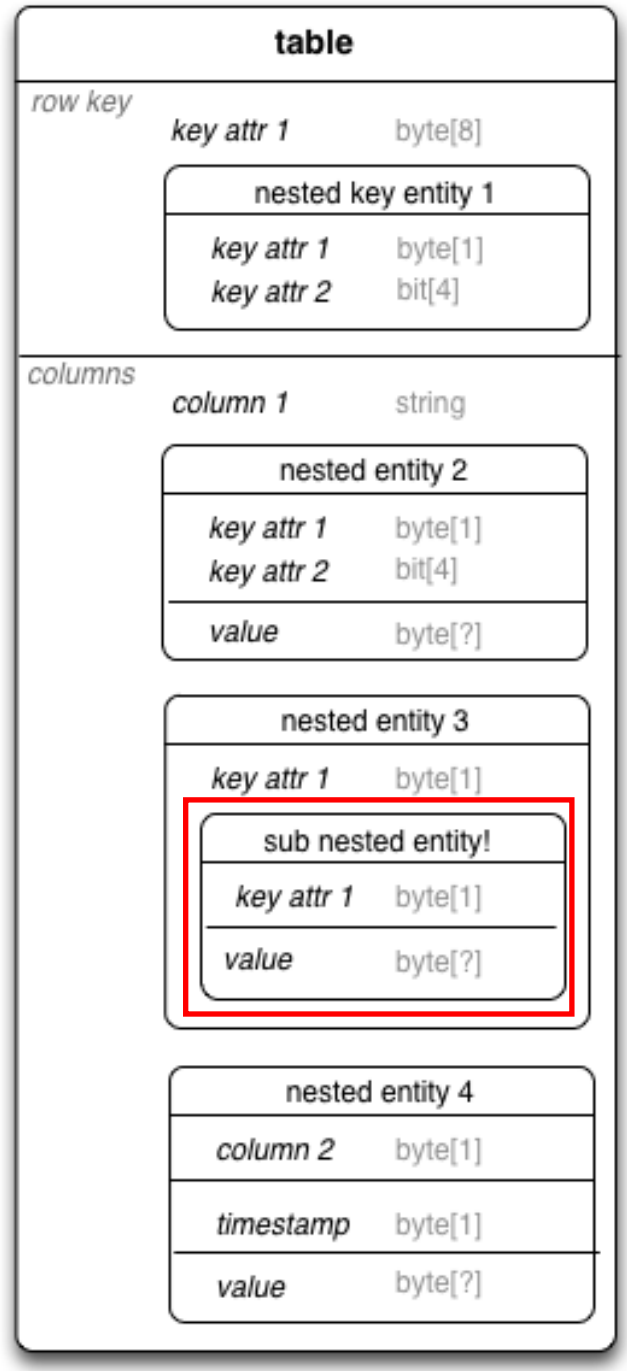
- In columns



Where can you nest entities?

Knock yourself out.

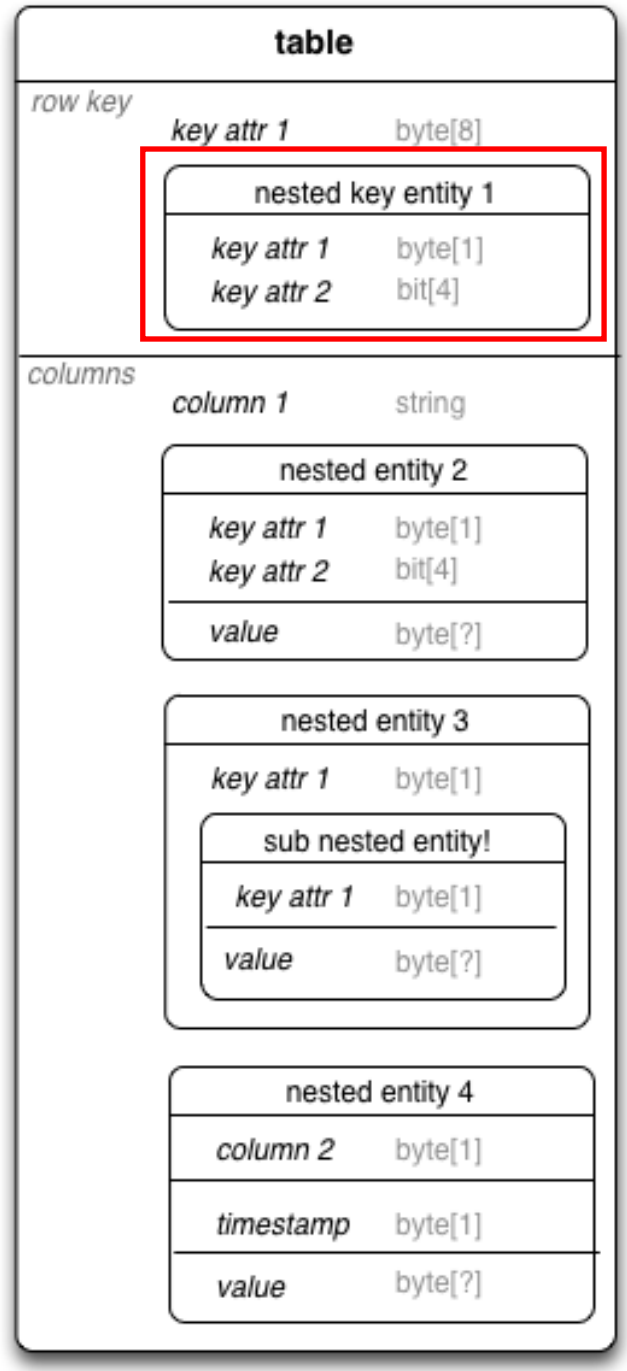
- In columns
- Two levels deep in columns!



Where can you nest entities?

Knock yourself out.

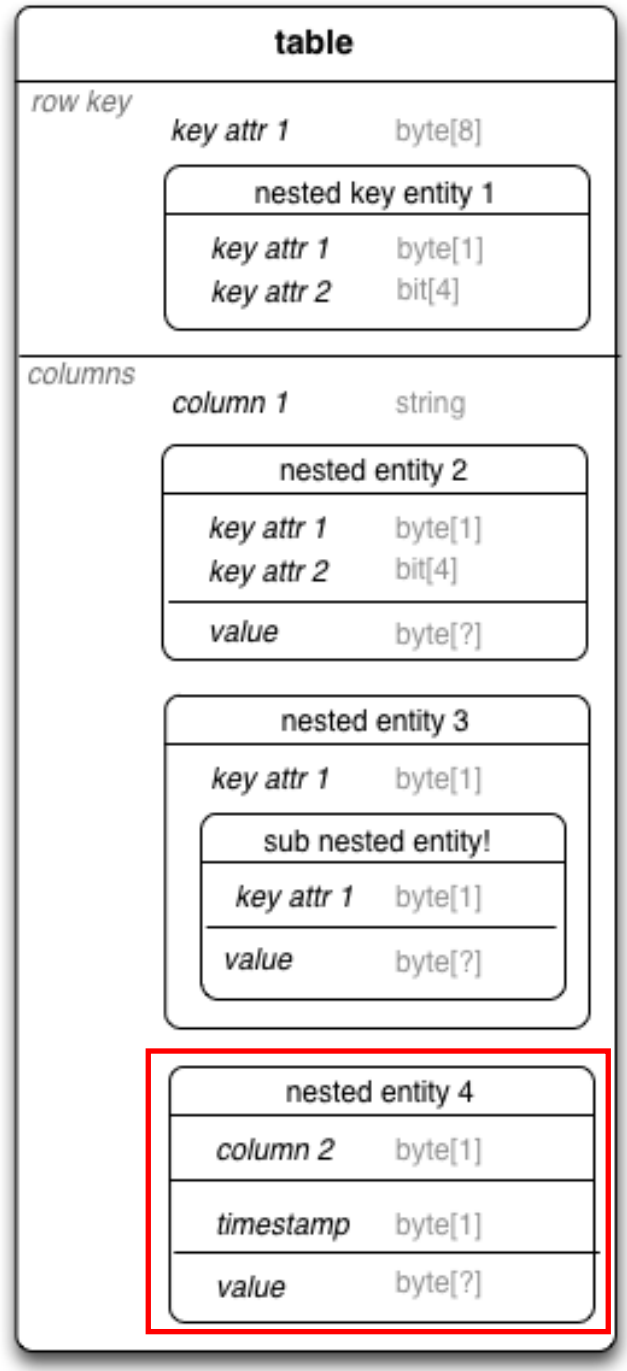
- In columns
- Two levels deep in columns!
- In the row key?!



Where can you nest entities?

Knock yourself out.

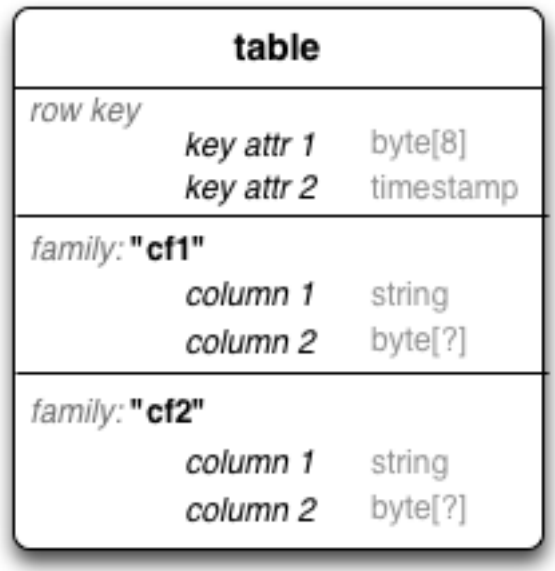
- In columns
- Two levels deep in columns!
- In the row key?!
- Using timestamps as a dimension!!



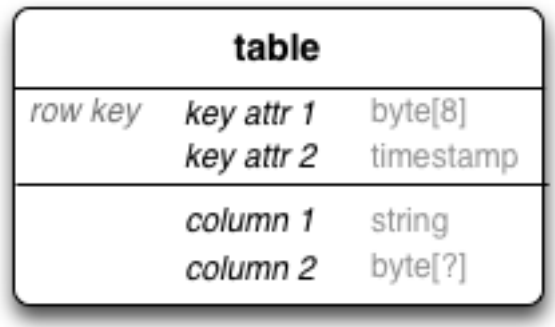
This is a fundamental modeling property of HBase: nesting entities.

Wait, what about Column Families?

They're just namespaces--additional vertical sections on the same entity.



Where column families aren't shown explicitly, let's assume there's just one.



So that brings us to a standard way to show an HBase schema:

Table: Top level entity name, fixed at design time.

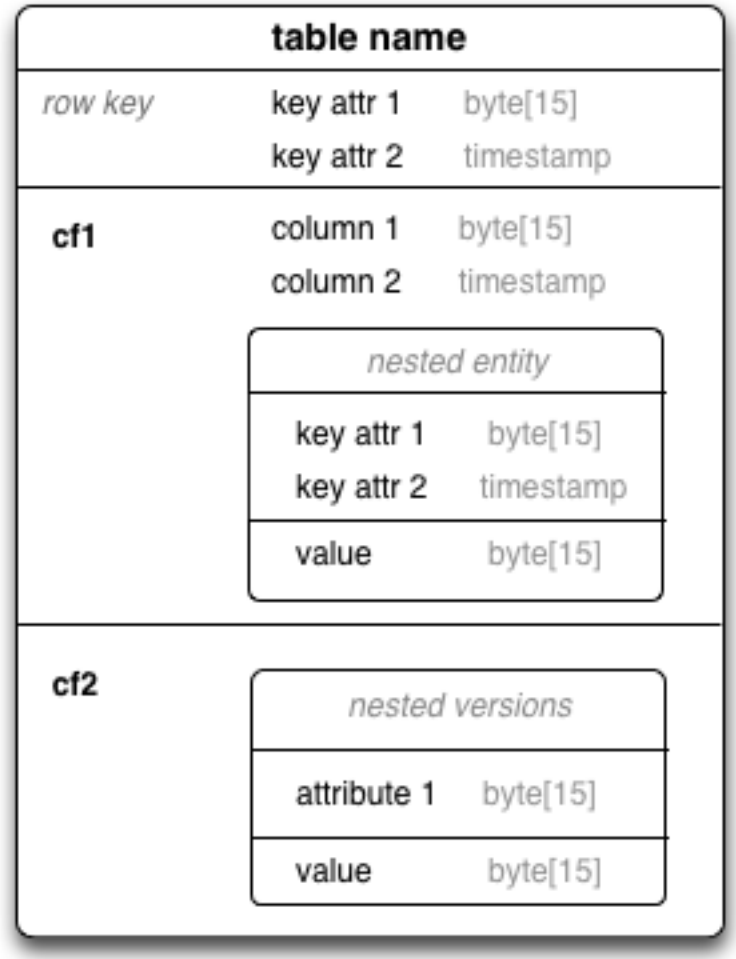
Row key: Can consist of multiple parts, each with a name & type. All data in one row shares the same row key. Can be nested.

Column family: A container that allows sets of columns to have separate storage & semantics.

Column qualifiers: Fixed attributes--design-time named columns, holding a value of a certain type.

Nested entities: Run-time named column qualifiers, where the qualifier is made up of one (or more) values, each with a type, along with one or more values (by timestamp) which also have a type.

Nested versions: Because multiple timestamped copies of each value can exist, you can also treat the timestamp as a modeled dimension in its own right. Hell, you could even use a long that's not a timestamp (like, say, an ID number). *Caveats apply.*



"But," you say, "what if I don't have all of these fancy modeling tools?"

Say it in text.

XML, JSON, DML, whatever you like.

```
<table name="Band">
  <key>
    <column name="band_id" type="int" />
  </key>
  <columnFamily name="cf1">
    <column name="band_name" type="string"/>
    <column name="hometown" type="string"/>
    <entity name="Show">
      <key>
        <column name="show_id">
      </key>
      <column name="date" type="date" />
    </entity>
  </columnFamily>
</table>
```

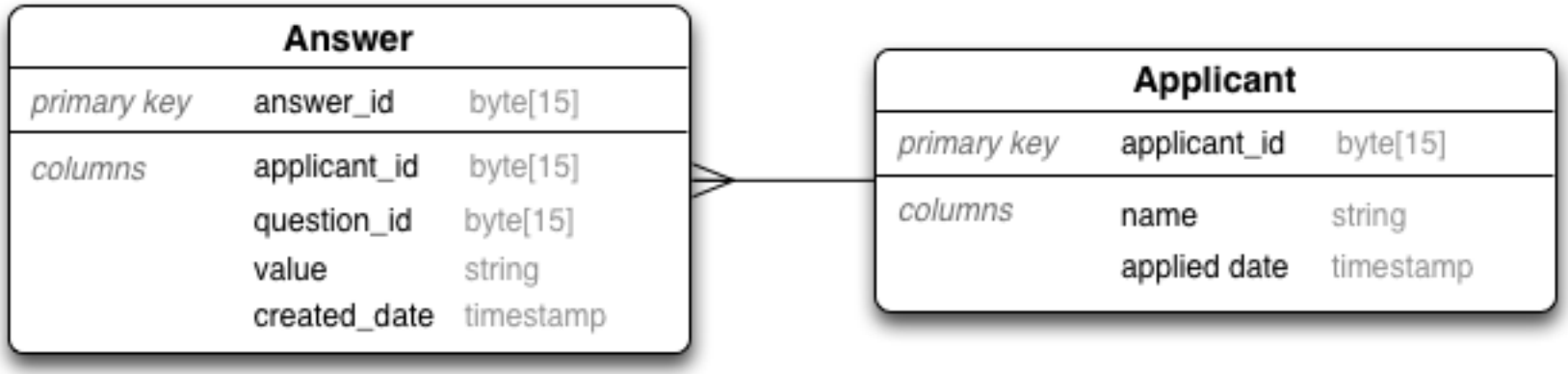
Text is faster and more general, but slightly harder to grok quickly.

So we'll stick with diagrams here.

Some examples are in order.

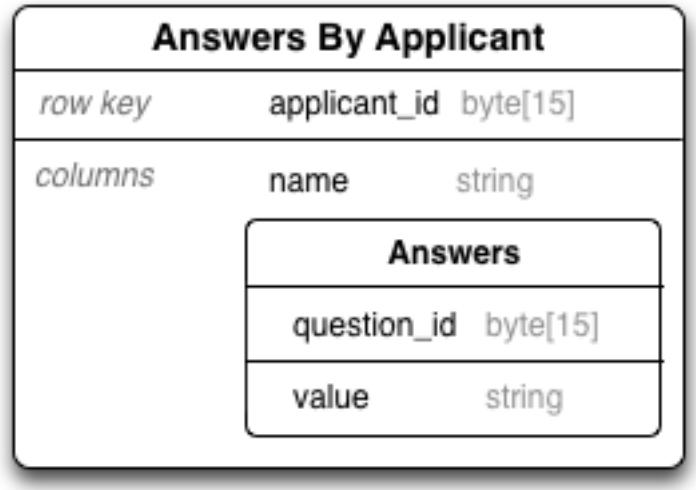
Example 1: a simple parent/child relationship

Relational Schema: **Applicants & Answers**



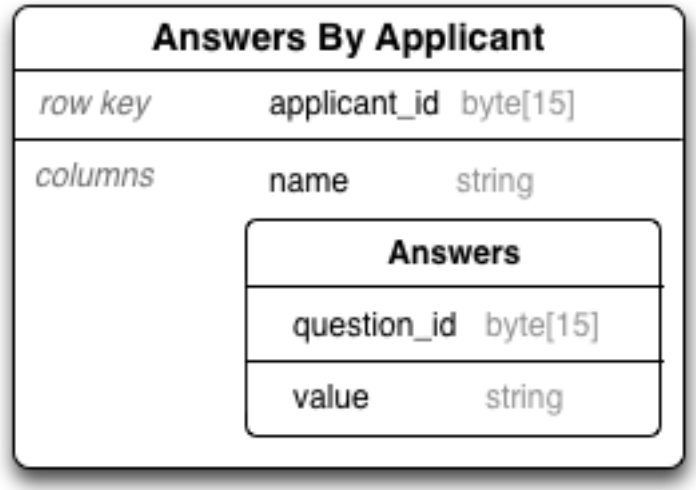
Standard parent/child relationship. One **Applicant** has many **Answers**; every **Answer** relates to a single **Applicant** (by id). It also relates to a **Question** (not shown). SQL would have you JOIN them to materialize an applicant and their answers.

HBase Schema: **Answers By Applicant**



- Answer is contained implicitly in Applicant
- If you know an applicant_id, you get O(1) access
- If you know an applicant_id AND question_id, O(1) access
- Answer.created_date is implicit in the timestamp on value

HBase Schema: **Answers By Applicant**



- You get answer history for free
- Applicant.applied_date can be implicit in the timestamp
- More attributes on applicant directly? Just add them!
- Answers are atomic and transactional by Applicant!!

Example of rows in HBase:

ap123	name = "John Smith", q1234="foo", q3456="bar"
ap456	name = "Tom Jones", q1234="baz"
ap789	name = "A Smithee", q2345="fred", q7584="flob", q9984="fee"

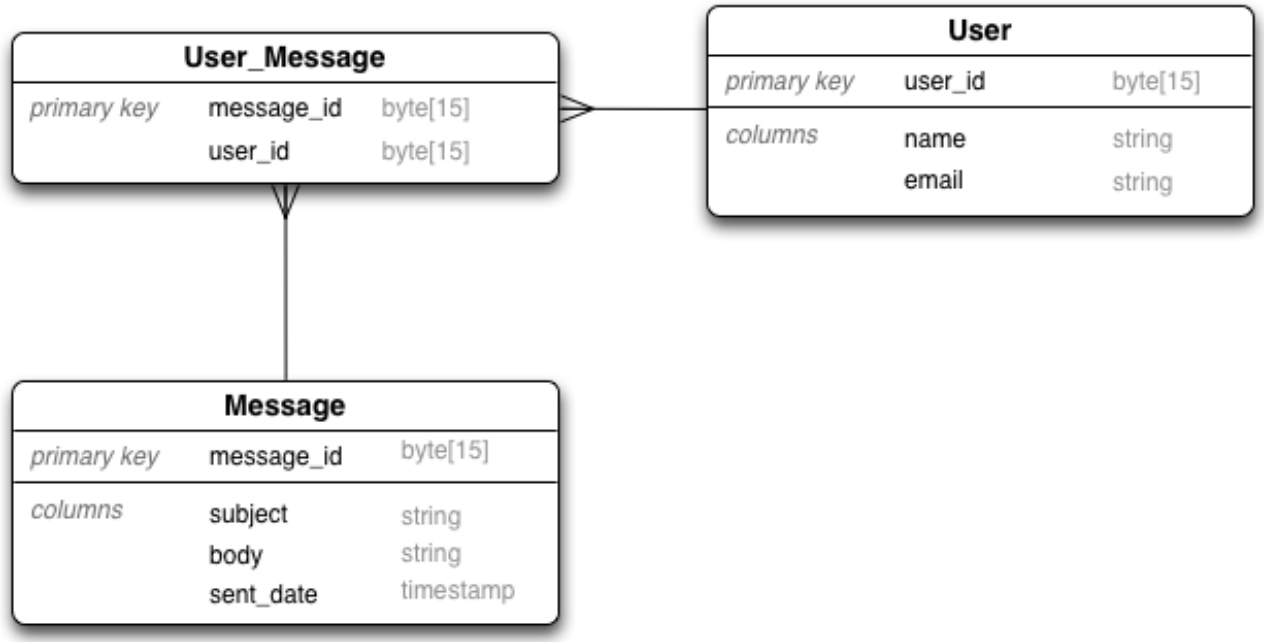
Before you get too excited, remember that there are cons to denormalization.

The cons:

- Nested entities aren't independent any more.
  - No: "SELECT avg(value) FROM Answer WHERE question_id = 123;"
  - But you can still do that with map/reduce
- This decomposition only works in one direction.
  - No relation chains without serious trickery.
  - Timestamps are another dimension, but that counts as trickery.
- No way to enforce the foreign key to another table.
  - But would you really do that in a big RDBMS?
- On disk, it repeats the row key for every column
  - You didn't really save anything by having applicant_id be implied.
  - Or did you? Compression negates that on disk ...
  - ... and prefix compression (HBASE-4218) will totally sink this.

Example 2: dropping some database science

Relational Schema: **Users And Messages**



Many-to-many relationship. One **User** sees many **Messages**, and a single **Message** can be seen by many **Users**. We want to do things like show the most recent message by subject (e.g. an inbox view).

What kind of SQL would you run on this? Say, we want to get the most recent 20 messages for 1 user.

```
SELECT TOP 20
  M.subject,
  M.body
  FROM
    User_Message UM
    INNER JOIN Message M
      ON UM.message_id = M.message_id
  WHERE
    UM.user_id = <user_id>
  ORDER BY
    M.sent_date DESC
```

Seems easy, right? Well, the database is doing some stuff behind the scenes for you:

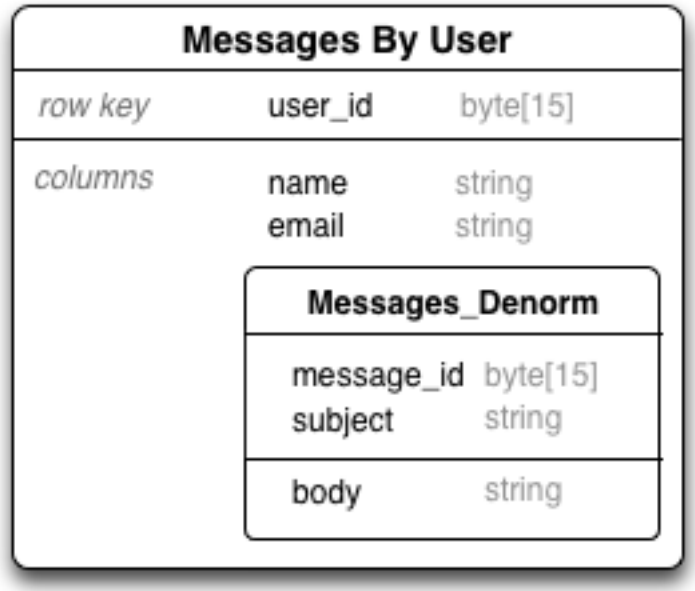
Assuming no secondary indexes, it might:

- Drive the join from the User_Message table
- For each new record with our given user_id, do a single disk access into the Message table (i.e. a hash_join)
- Get the records for *every* message for this user
- Sort them all
- Take the top 20

No shortcuts; we can't find the top 20 by date w/o seeing ALL messages.

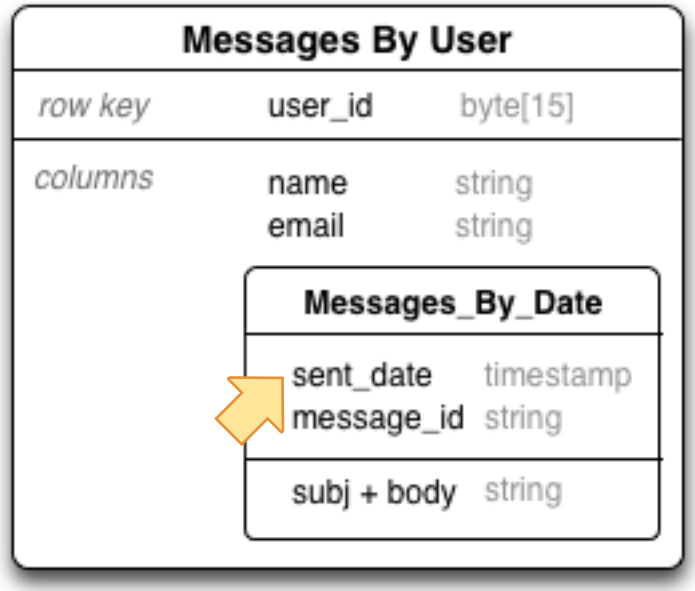
This gets more expensive as a user gets more messages. (But it's still pretty fast if a given user has a reasonable number of messages).

How could you do this in HBase?
Try the same pattern as parent / child?



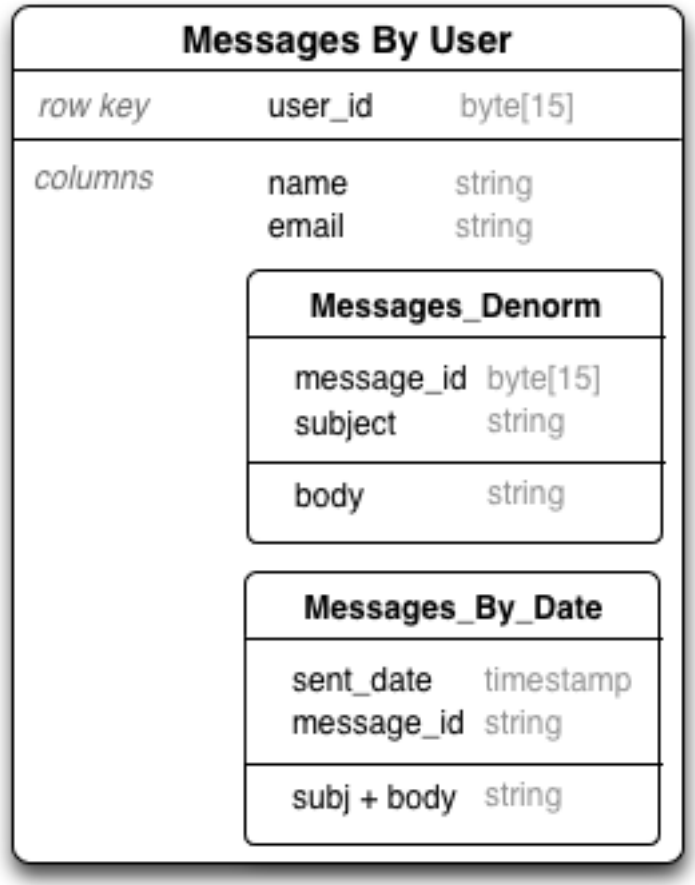
1. Because of the many-to-many, you now have N copies of the message (one per user).
    - Maybe that's OK (especially if it's immutable!). Disk is cheap.
2. Your statement has to do the same amount of work, but now you have to do it yourself. :(

If I **know** that I always want it ordered by date, why not store it that way?



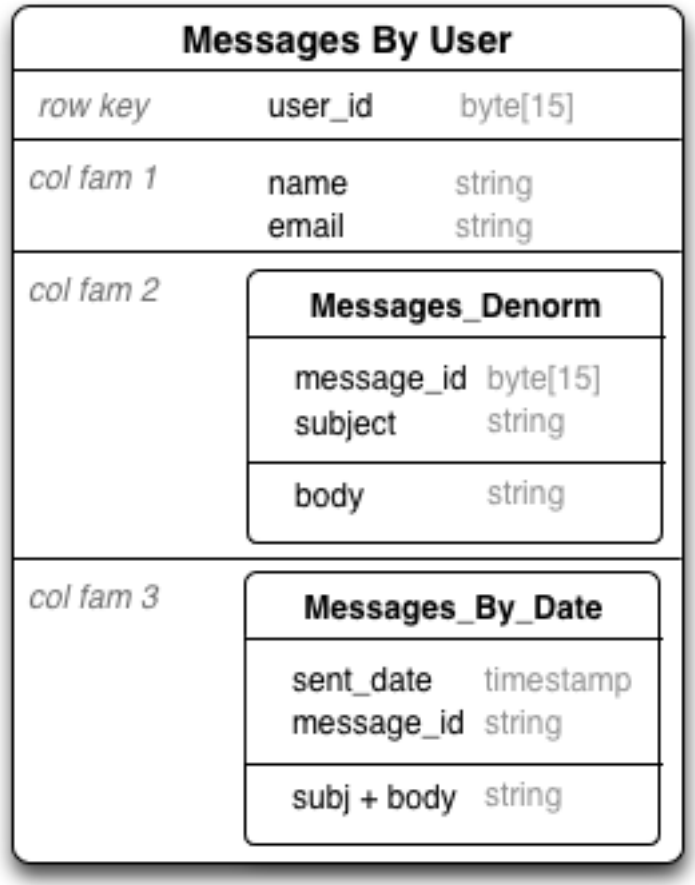
- Now I can scan over messages by date until I get enough; it's O(1)
- But what if I want it by message_id again? Doh. I'm screwed, unless ...

I store it both ways!



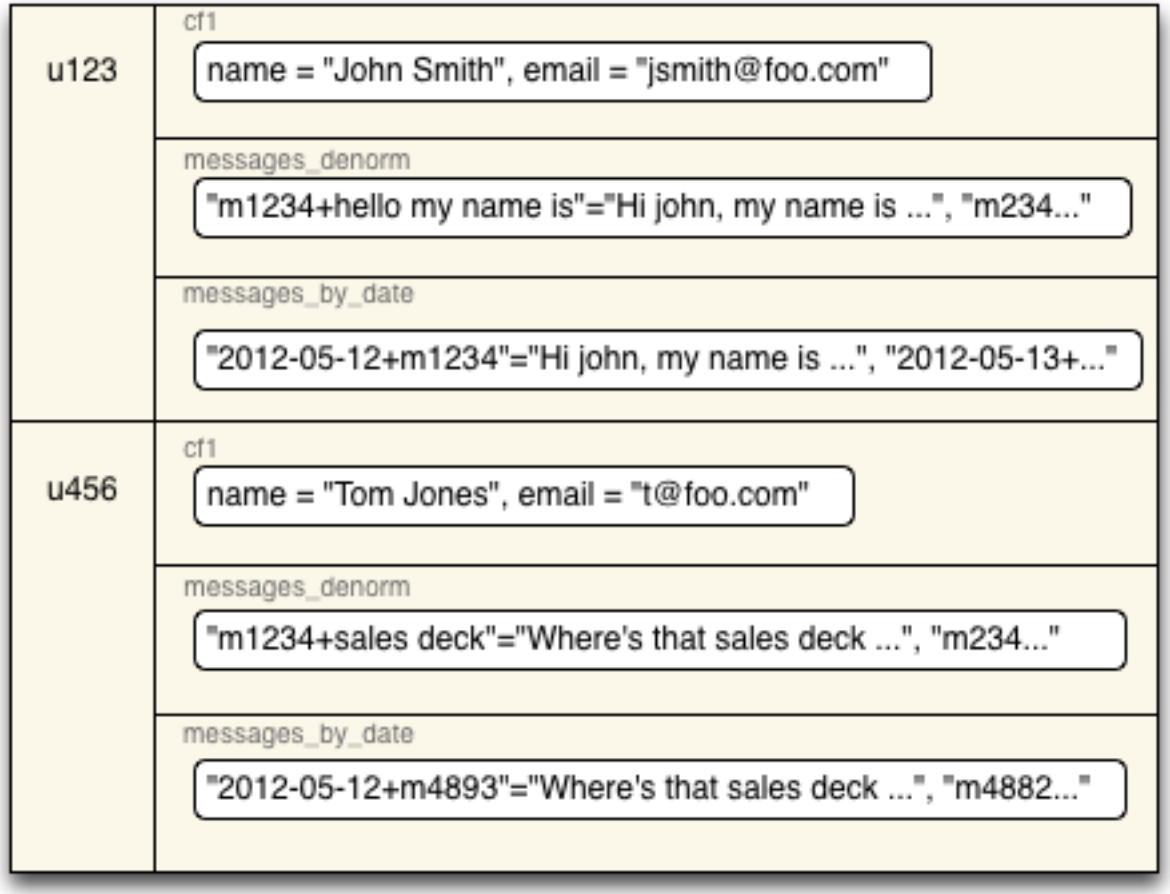
Nice: updates to this are transactional (consistent) for a given user, because it's all in one row. So it's not a bear to maintain.

Which I could even do in different column families ...

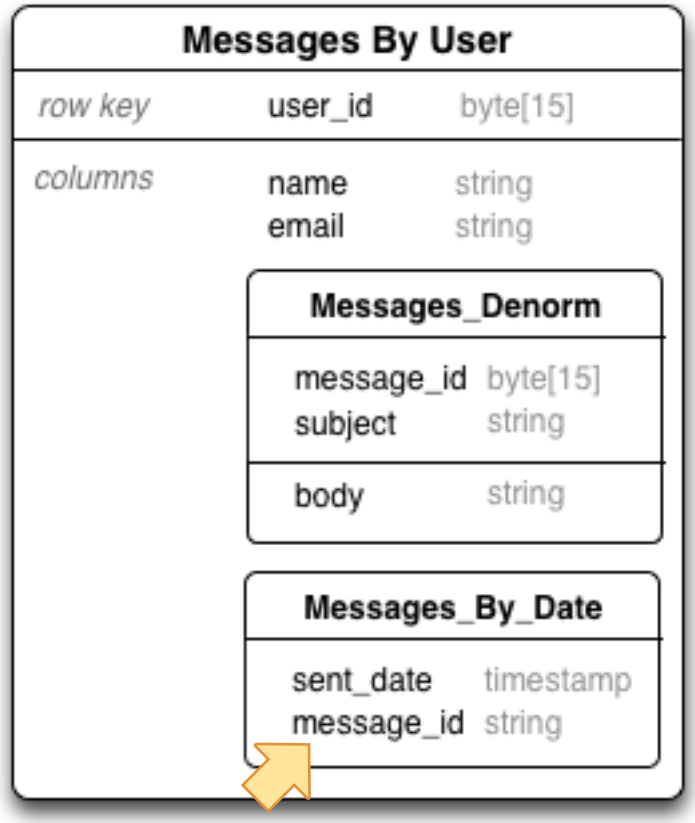


(Makes sense if I don't usually access them at the same time; I only pay the cost for the query I am running.)

So, for example ...



Or I could just use the by-date one as an "index" ...



So I only store the subject and body once.
This means I need to perform my own "join" in code.

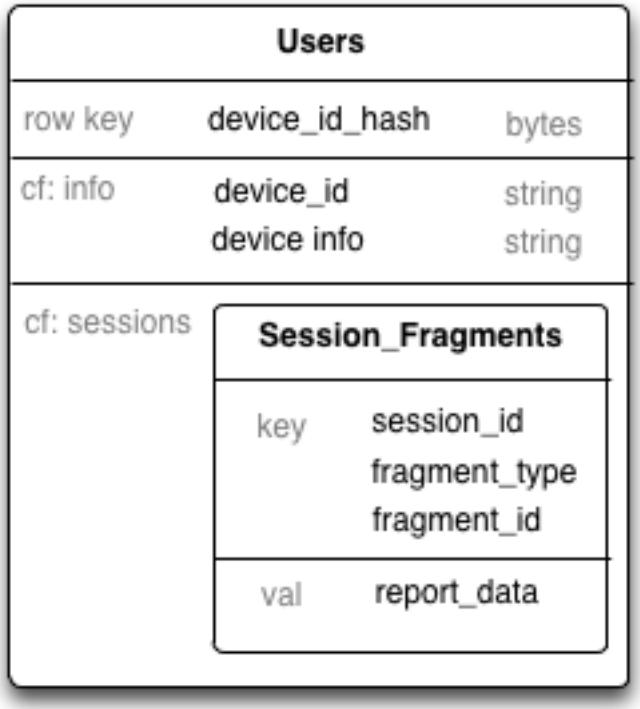
See a theme emerging?

Relational DBs lull you into **not** thinking about physical access as much; but when you have to scale, there are hard limits.

HBase makes you think about it sooner, but gives you the tools to think about it in more a straightforward way.

Example 3: Flurry

See: http://www.flurry.com/data/

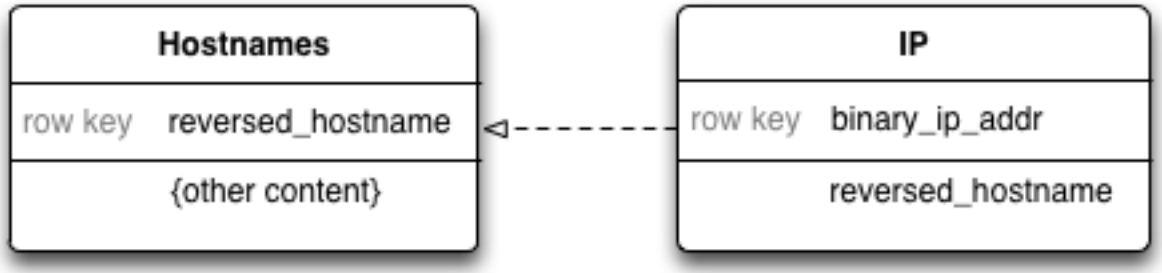


- One row per device
- Nested entity in CF "Sessions"
  - Polymorphic "fragments" of session reports
  - Map/Reduce transforms

Caveat: this is based on email exchanges, so the details may be wonky, but I think the overall point is right.

Example 4: Indexes

Example from Andrew Purtell



- Using one table as an index into another often makes sense
- Can't be transactional (without external coordination)
- So you have to keep it clean, or tolerate dirty indexes
- Note that there are several attempts to add solid general purpose indexing to HBase, but so far none have caught on.

Same caveat: this is based on email exchanges, so the details may be wonky, but I think the overall point is right.

Here are some more design patterns.

0: The row key design is the single most important decision you will make.

0: The row key design is the single most important decision you will make.

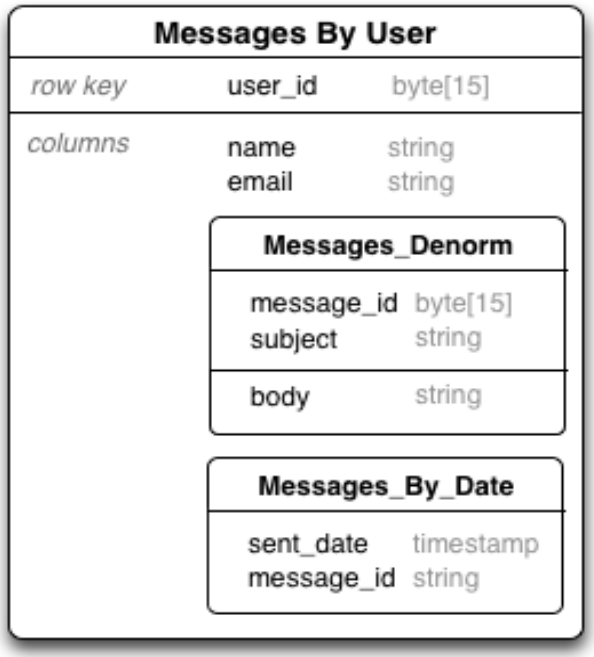
This is also true for the "key" you're putting in the column family name of nested entities.

1: Design for the questions,
not the answers.

1: Design for the questions, not the answers.

(hat tip to Ilya Katsov from the High Scalability blog for this useful way to put it; and possibly to Billy Newport or Eben Hewitt for saying it first.)

1: Design for the questions, not the answers.



Let's be clear: this sucks big time, if you aren't 100% sure what the questions are going to be.

Let's be clear: this sucks big time, if you aren't 100% sure what the questions are going to be.

Use a relational DB for that!

Or a document database like CouchDB ...

"But isn't NoSQL more flexible than a relational DB?"

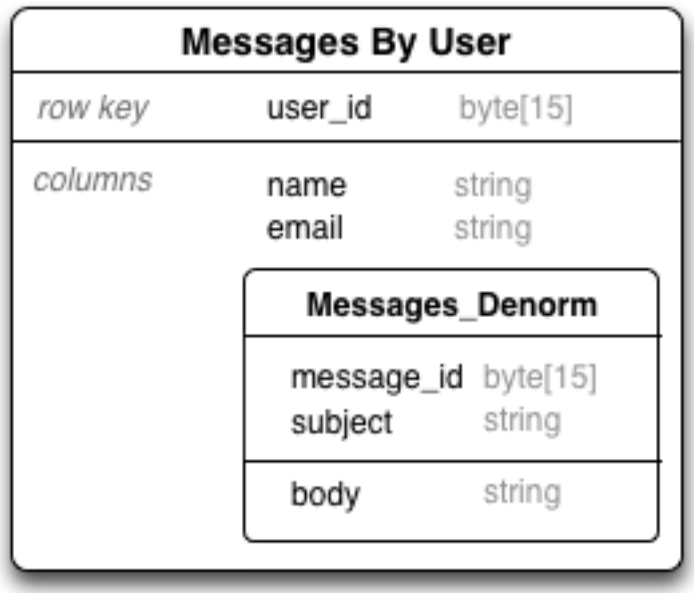
For column schema? Yes!
For row key structures, NO!

2: There are only two sizes of data:
too big, and not too big.

2: There are only two sizes of data:
too big, and not too big.

(That is, too big to scan all of something
while answering an interactive request.)

2: There are only two sizes of data: too big, and not too big.

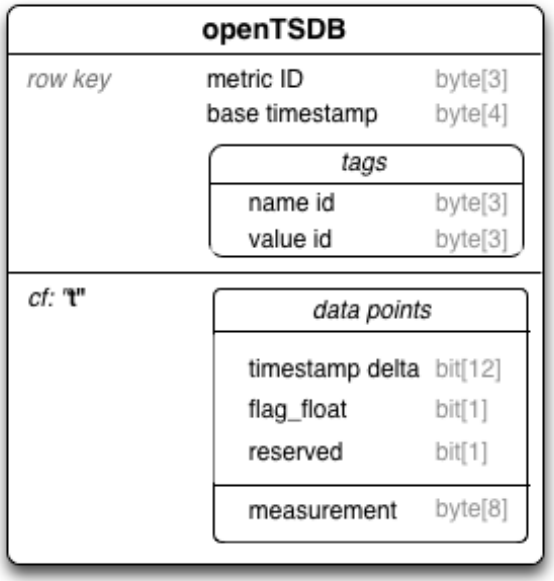


3: Be compact.

You can squeeze a lot into a little space.

3: Be compact.

You can squeeze a lot into a little space.



This is also important because the rowkey and CF name are repeated for every single value (memory and disk).

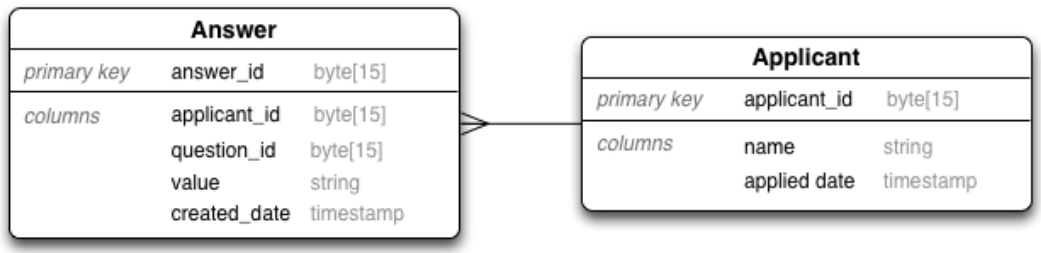
File compression can negate this on disk, and prefix compression will probably negate this in memory.

4: Use row atomicity as a design tool.

Rows are updated atomically, which gives you a form of relational integrity in HBase!

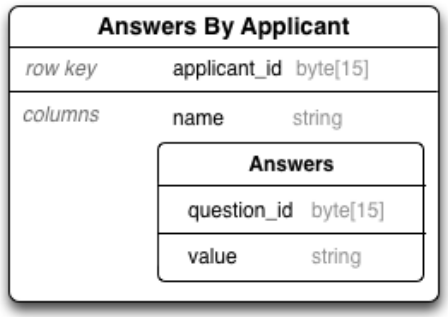
4: Use row atomicity as a design tool.

If you made this two HBase tables, you couldn't guarantee integrity (updated to one could succeed, while updates to the other fail).



4: Use row atomicity as a design tool.

If you make it one table with a nested entity, you can guarantee updates will be atomic, and you can do much more complex mutations of state.

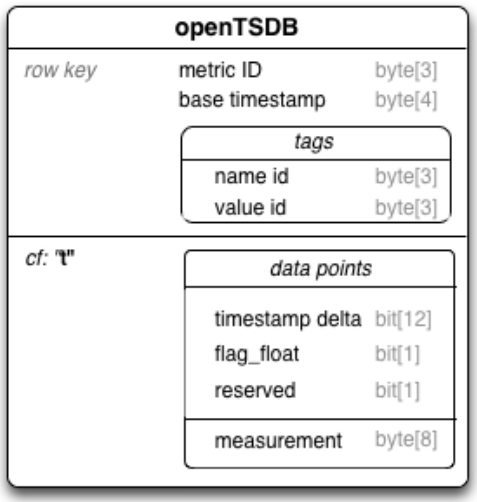


5: Attributes can move into the row key

Even if it's not "identifying" (part of the uniqueness of an entity), adding an attribute into the row key can make access more efficient in some cases.

5: Attributes can move into the row key

Even if it's not "identifying" (part of the uniqueness of an entity), adding an attribute into the row key can make access more efficient in some cases.



5: Attributes can move into the row key

This ability to move things left or right (without changing the physical storage requirements) is part of what Lars George calls "folding".

Also, if you like this subject, go watch his videos on advanced HBase schema design, they're awesome.

6: If you nest entities, you can transactionally pre-aggregate data.

You can recalculate aggregates on write,
or periodically with a map/reduce job.

Practically: how do you load schema into HBase?

Practically: how do you load schema into HBase?

Direct actuation: https://github.com/larsgeorge/hbase-schema-manager

Practically: how do you load schema into HBase?

Direct actuation: https://github.com/larsgeorge/hbase-schema-manager
Coming soon: script generation: https://github.com/ivarley/scoot

Thank you!

Questions? @thefutureian