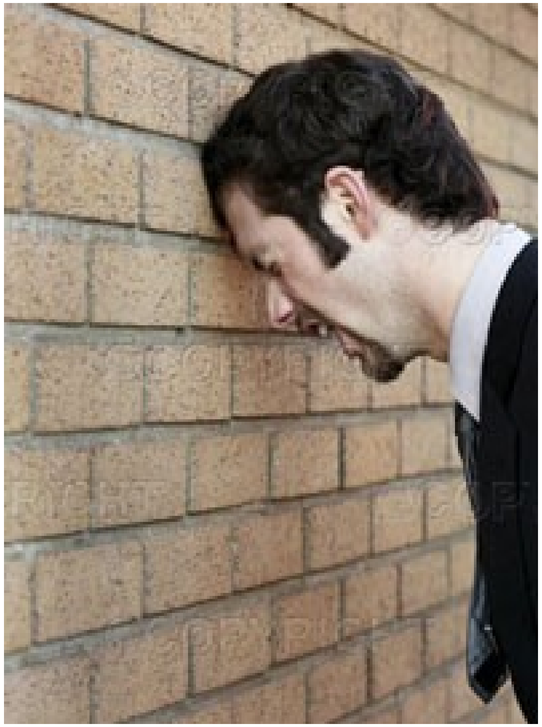
Simplifying Database Design

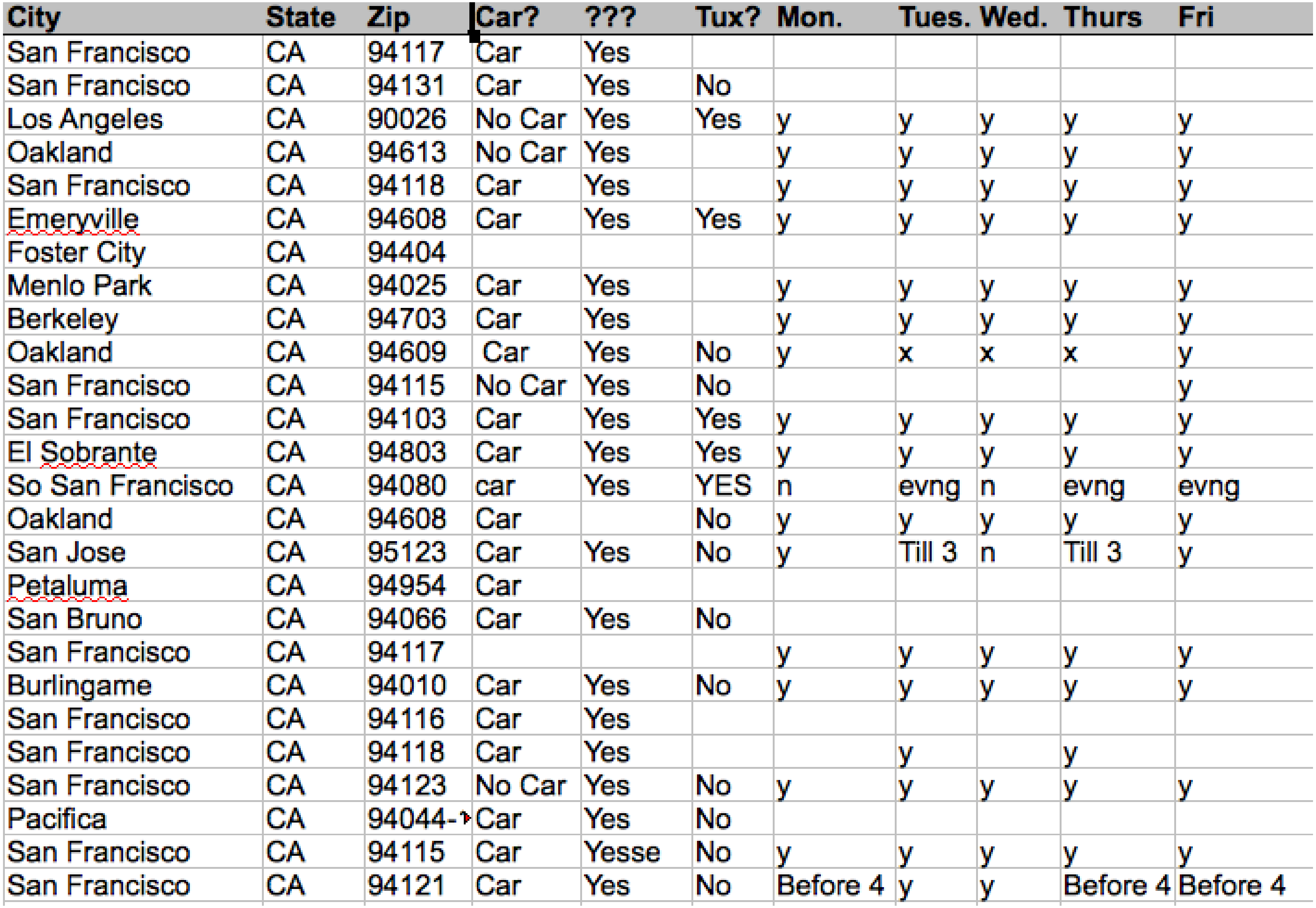
Josh Berkus
PostgreSQL Experts, Inc.
O'Reilly OSCON 2009



How Not To Do It

four popular methods

1. One Big Spreadsheet



City	State	Zip	Car?	???	Tux?	Mon.	Tues.	Wed.	Thurs	Fri
San Francisco	CA	94117	Car	Yes						
San Francisco	CA	94131	Car	Yes	No					
Los Angeles	CA	90026	No Car	Yes	Yes	y	y	y	y	y
Oakland	CA	94613	No Car	Yes		y	y	y	y	y
San Francisco	CA	94118	Car	Yes		y	y	y	y	y
Emeryville	CA	94608	Car	Yes	Yes	y	y	y	y	y
Foster City	CA	94404								
Menlo Park	CA	94025	Car	Yes		y	y	y	y	y
Berkeley	CA	94703	Car	Yes		y	y	y	y	y
Oakland	CA	94609	Car	Yes	No	y	x	x	x	y
San Francisco	CA	94115	No Car	Yes	No					y
San Francisco	CA	94103	Car	Yes	Yes	y	y	y	y	y
El Sobrante	CA	94803	Car	Yes	Yes	y	y	y	y	y
So San Francisco	CA	94080	car	Yes	YES	n	evng	n	evng	evng
Oakland	CA	94608	Car		No	y	y	y	y	y
San Jose	CA	95123	Car	Yes	No	y	Till 3	n	Till 3	y
Petaluma	CA	94954	Car							
San Bruno	CA	94066	Car	Yes	No					
San Francisco	CA	94117				y	y	y	y	y
Burlingame	CA	94010	Car	Yes	No	y	y	y	y	y
San Francisco	CA	94116	Car	Yes						
San Francisco	CA	94118	Car	Yes			y		y	
San Francisco	CA	94123	No Car	Yes	No	y	y	y	y	y
Pacifica	CA	94044-	Car	Yes	No					
San Francisco	CA	94115	Car	Yesse	No	y	y	y	y	y
San Francisco	CA	94121	Car	Yes	No	Before 4	y	y	Before 4	Before 4

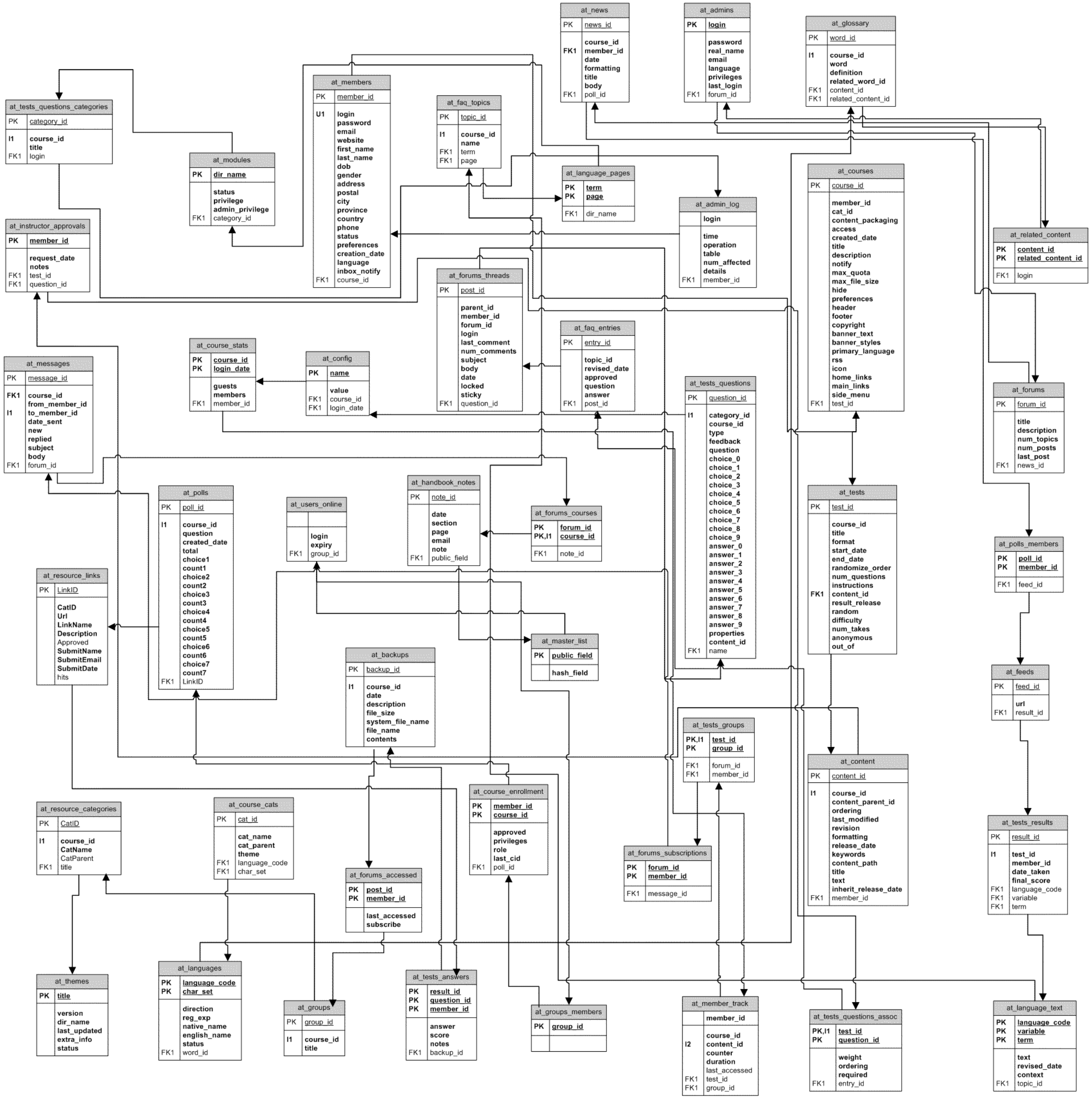
2. Hashes, EAV & E-Blob

ID	Property	Setting
407	Eyes	Brown
407	Height	73in
407	Married?	TRUE
408	Married?	FALSE
408	Smoker	FALSE
408	Age	37
409	Height	66in

ID	Properties
407	<eyes=”brown”><height=”73”> <married=”1”><smoker=”1”>
408	<hair=”brown”><age=”49”> <married=”0”><smoker=”0”>
409	<age=”37”><height=”66”> <hat=”old”><teeth=”gold”>

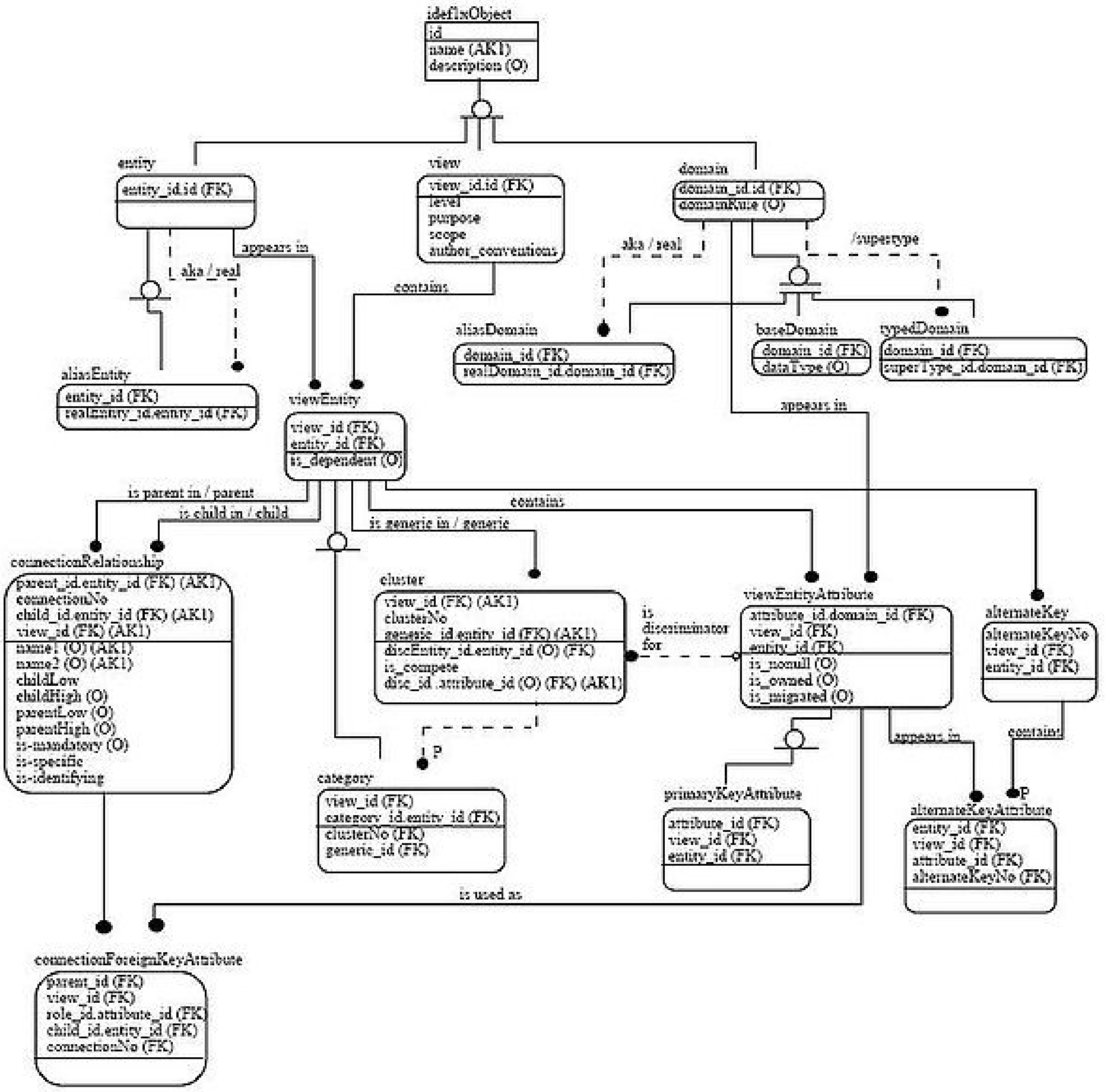
3. Incremental Development

4. Leave It to the ORM

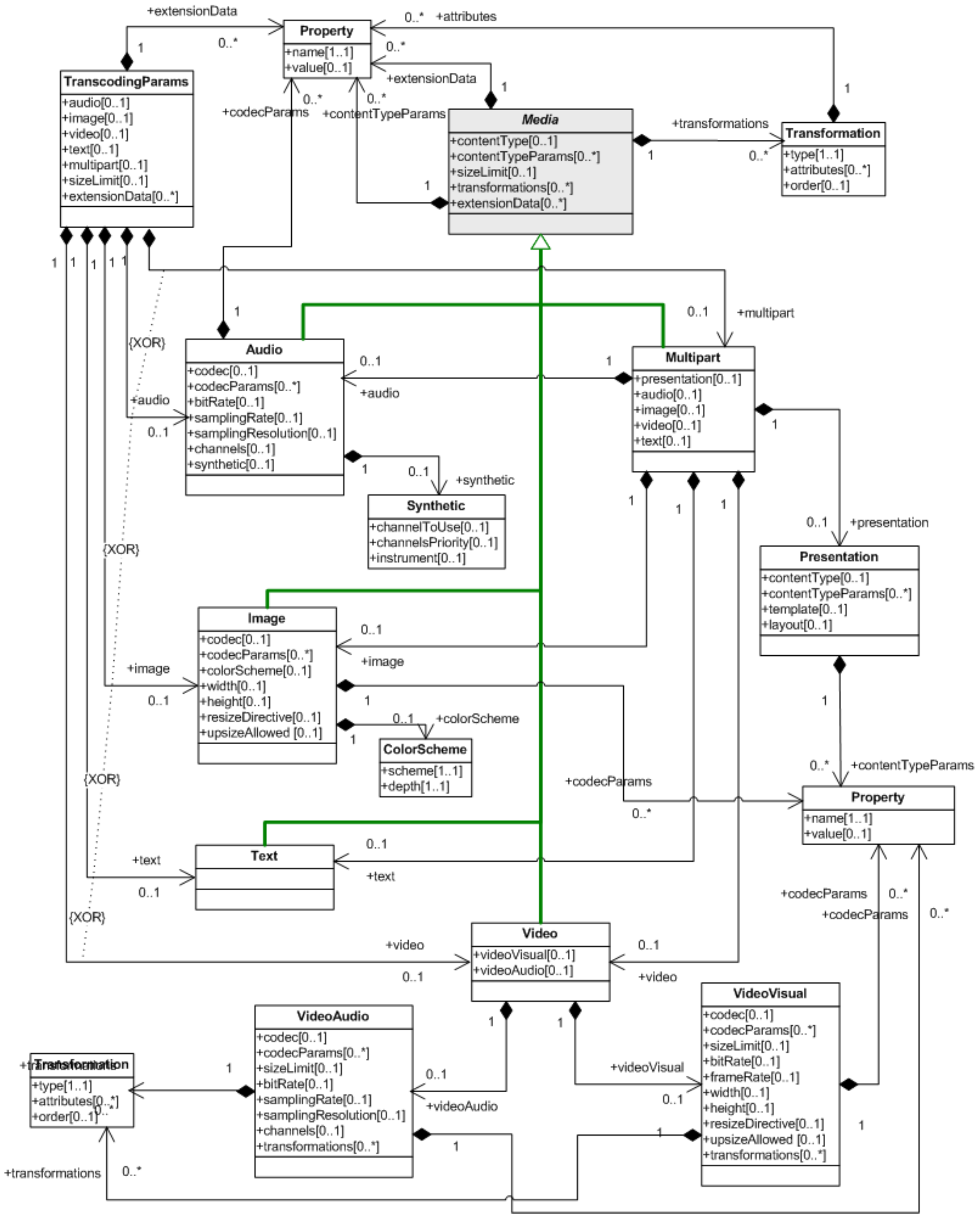


Data Modeling

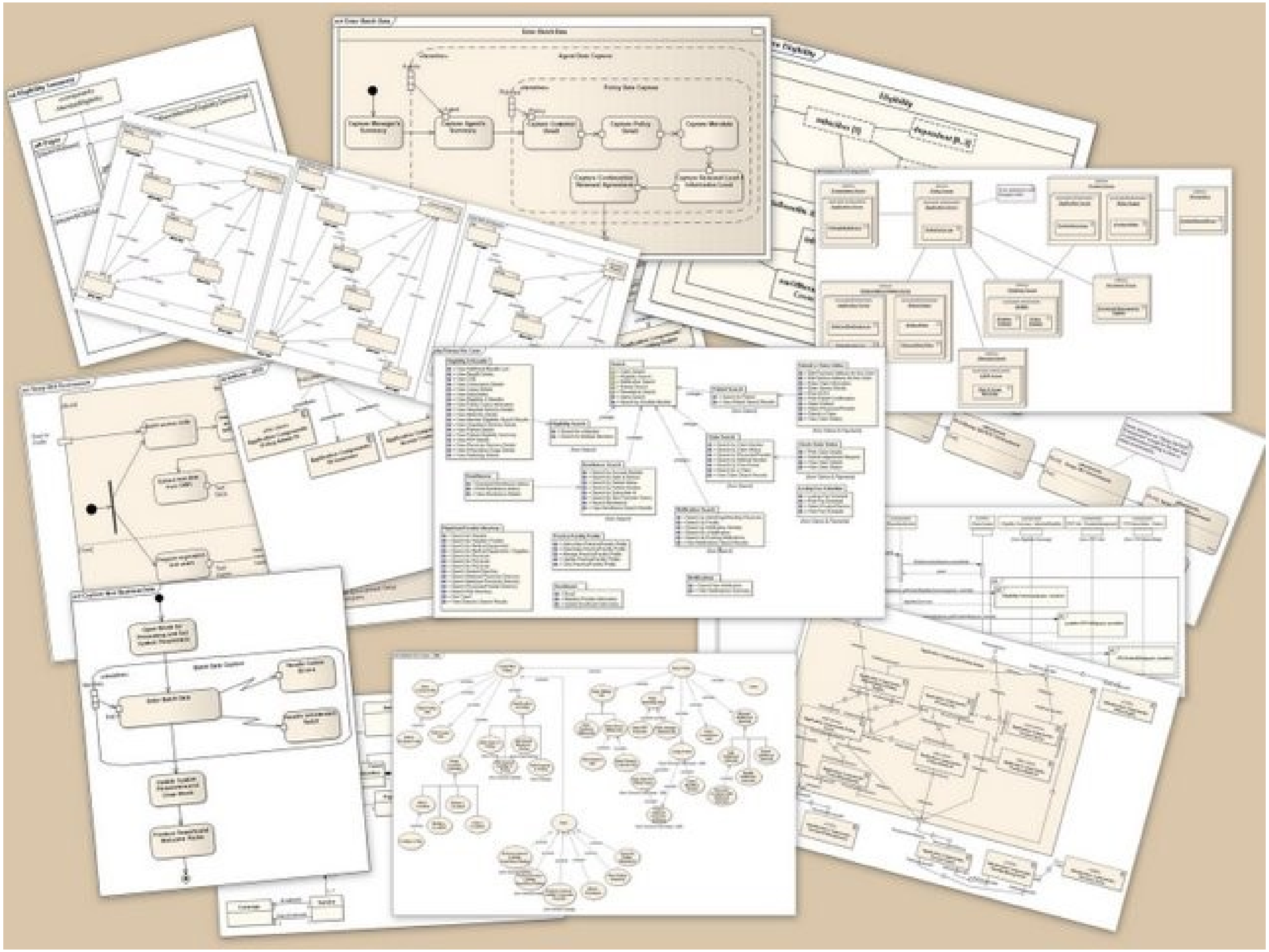
Entity Resource Diagram



Unified Modeling Language



Wait, which standard?



Simple Bulletin Board Database Design

Database As Model

1. Your database is a model of your application

2. Your application is a model of your problem domain

conclusion: you can *simply* model the database as a derivative of your problem domain

corollary: if you don't understand your database, you don't understand the problem you're solving

Get Together
Your *Whole* Dev Team

Why the *whole* team?

- You need to know the *entire* problem you're modeling through the database.
- Some developers may be working on specific features which need database support which the managers forget about.
- All developers need to understand that the database is part of the software development and release cycle.

Start with a List *“things” we need to store*

- Forums
- Threads
- Posts
- Users
- Administrators
- Messages

Simple Relationships

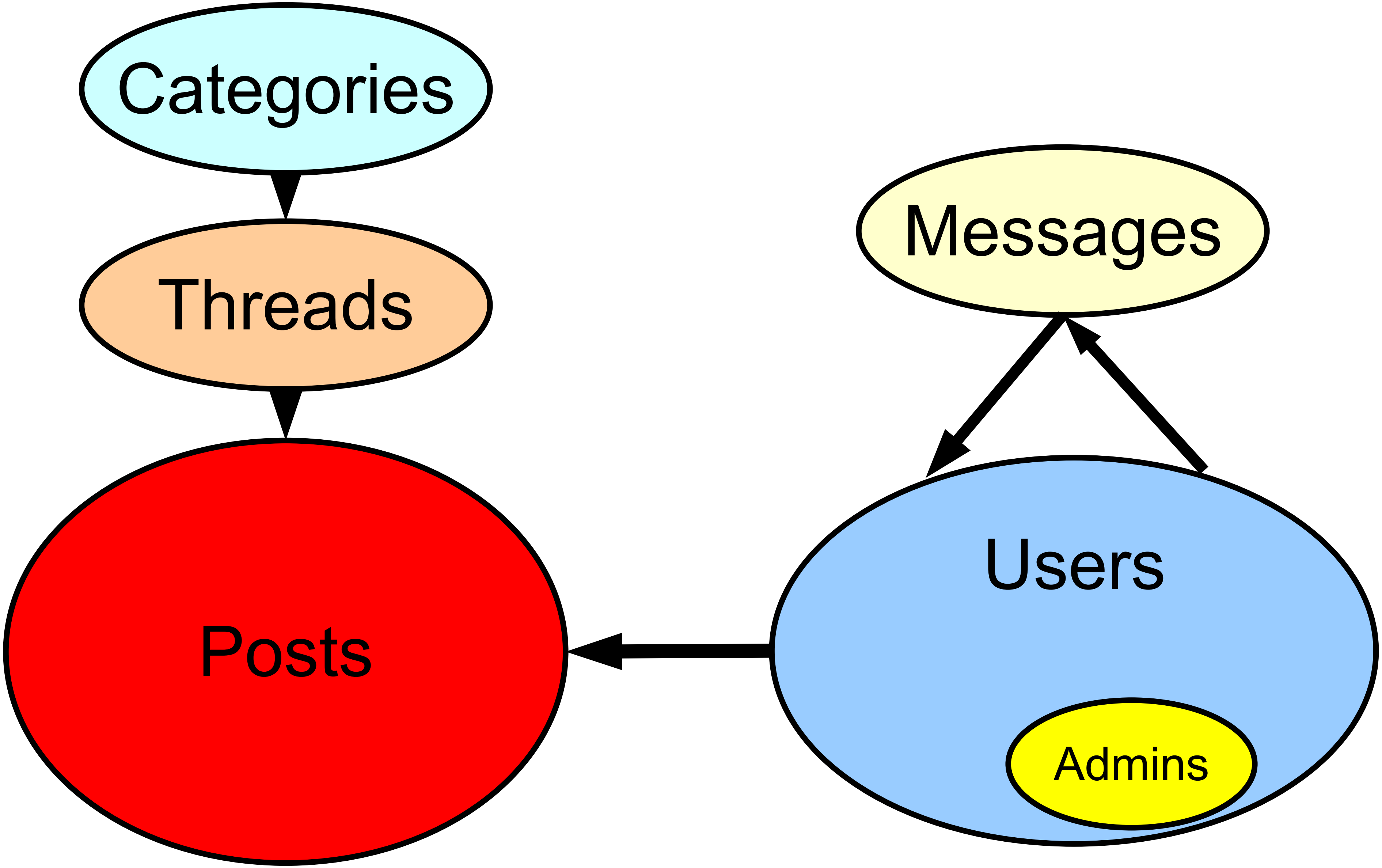


Figure out the Attributes of each “thing”

- name
- email
- login
- password
- status

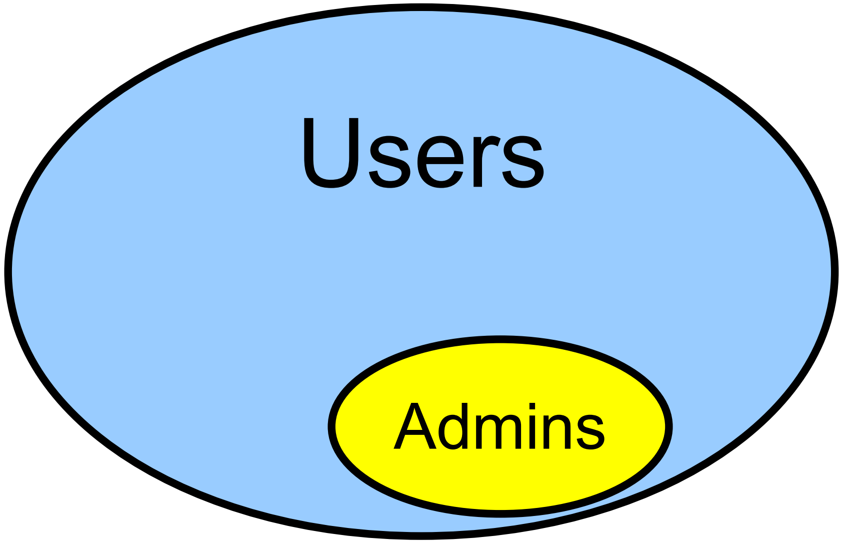
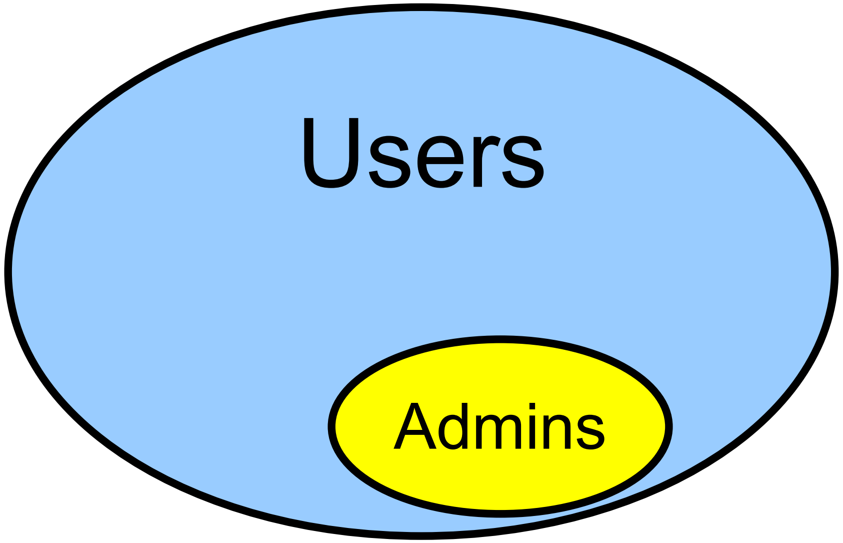


Figure out what kind of data

name	text
email	text – special
login	text
password	text
status	char

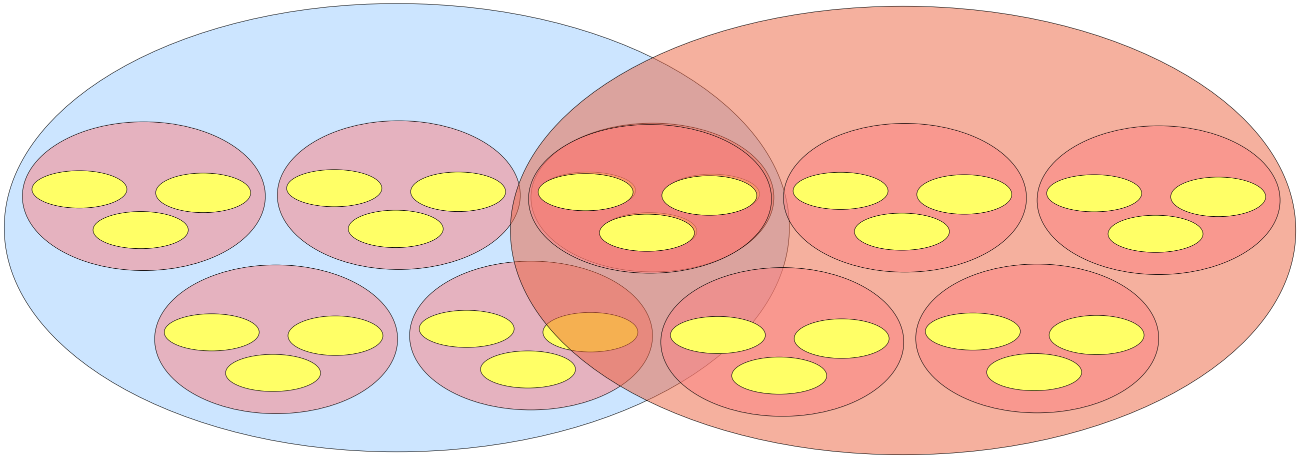


Repeat for all “Things”

- forums
  - name, description, owner, created
- threads
  - name, description, owner, created
- posts
  - created, owner, content, flag
- messages
  - sender, recipients, subject, content

OK, Now Get Out!

Interlude



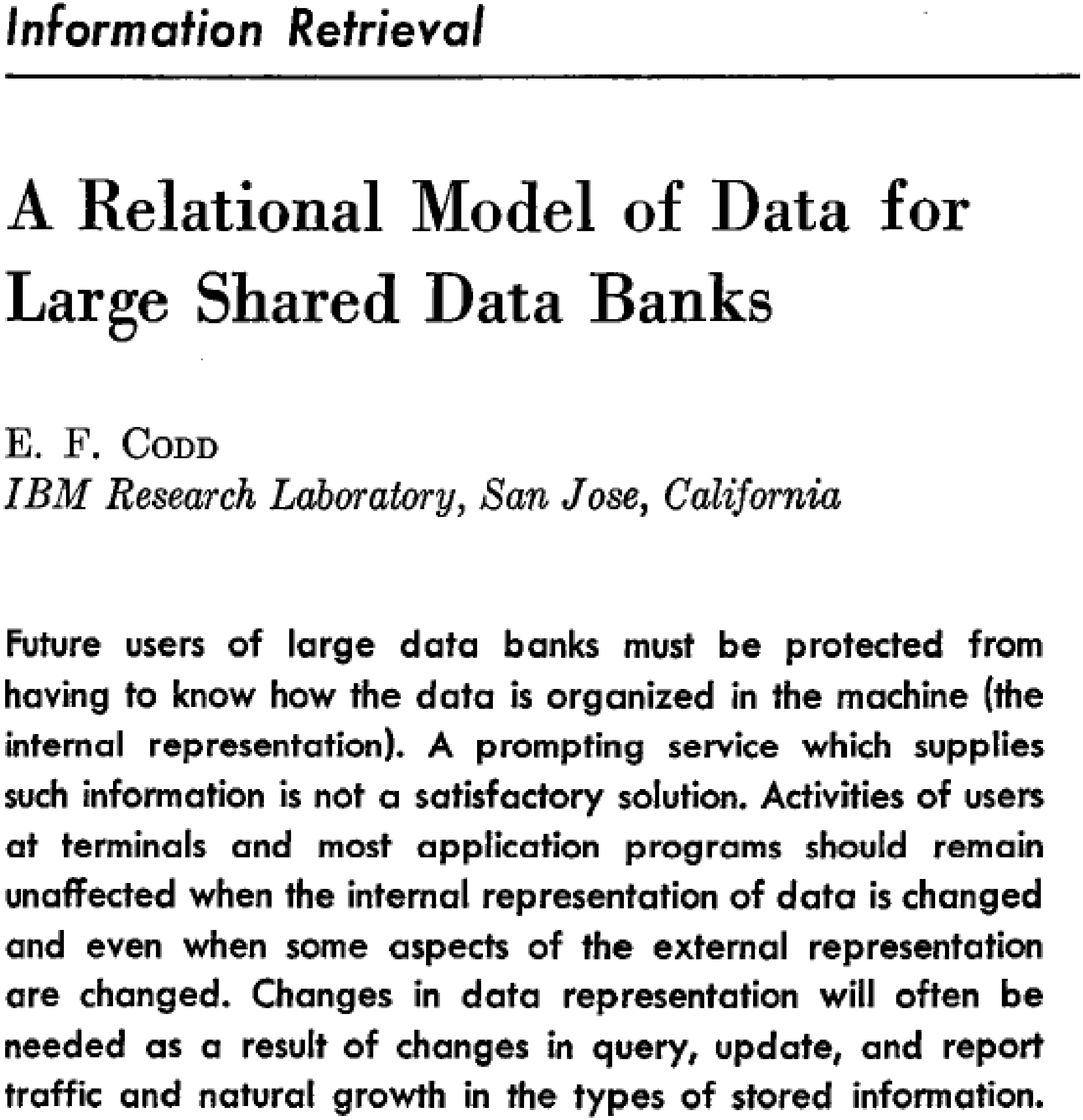
All
the Relational Theory
You Need to Know
in 20 Minutes

E.F. Codd
Database Engineer, IBM 1970

IBM Databases Run Amok

1. losing data
2. duplicate data
3. wrong data
4. crappy performance

5. downtime for database redesign whenever anyone made an application change



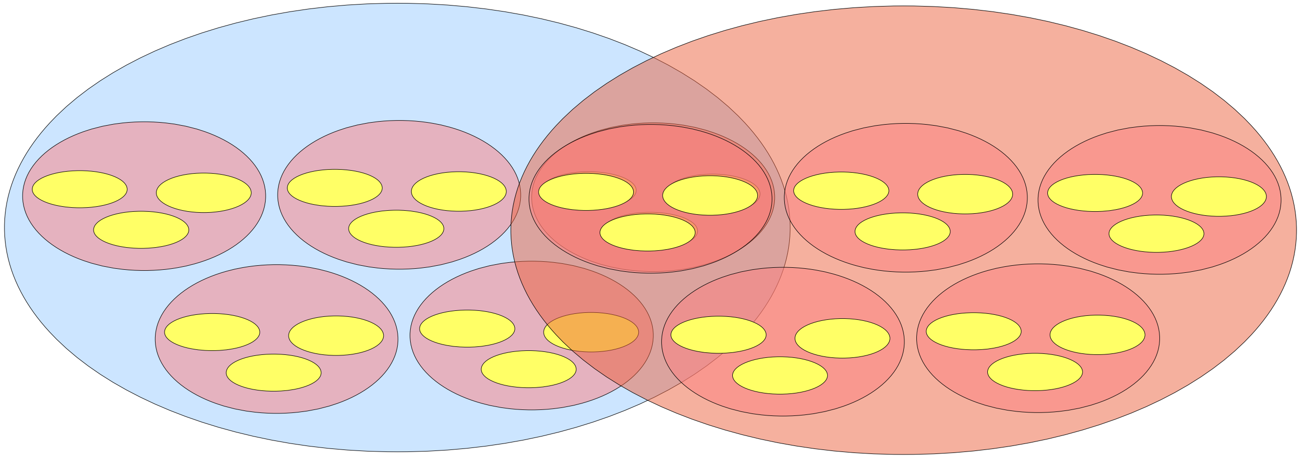
Information Retrieval

A Relational Model of Data for Large Shared Data Banks

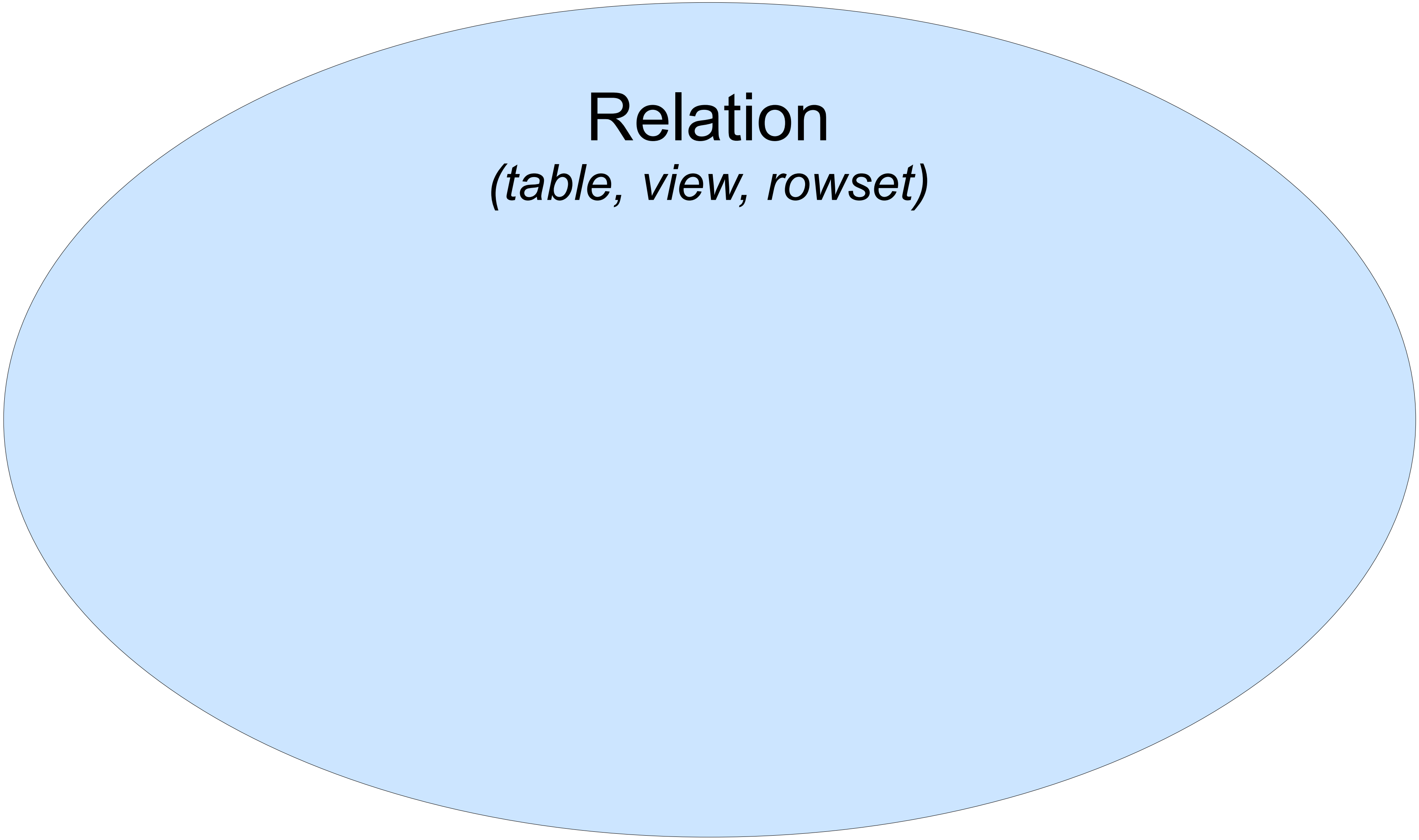
E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Set (Bag) Theory



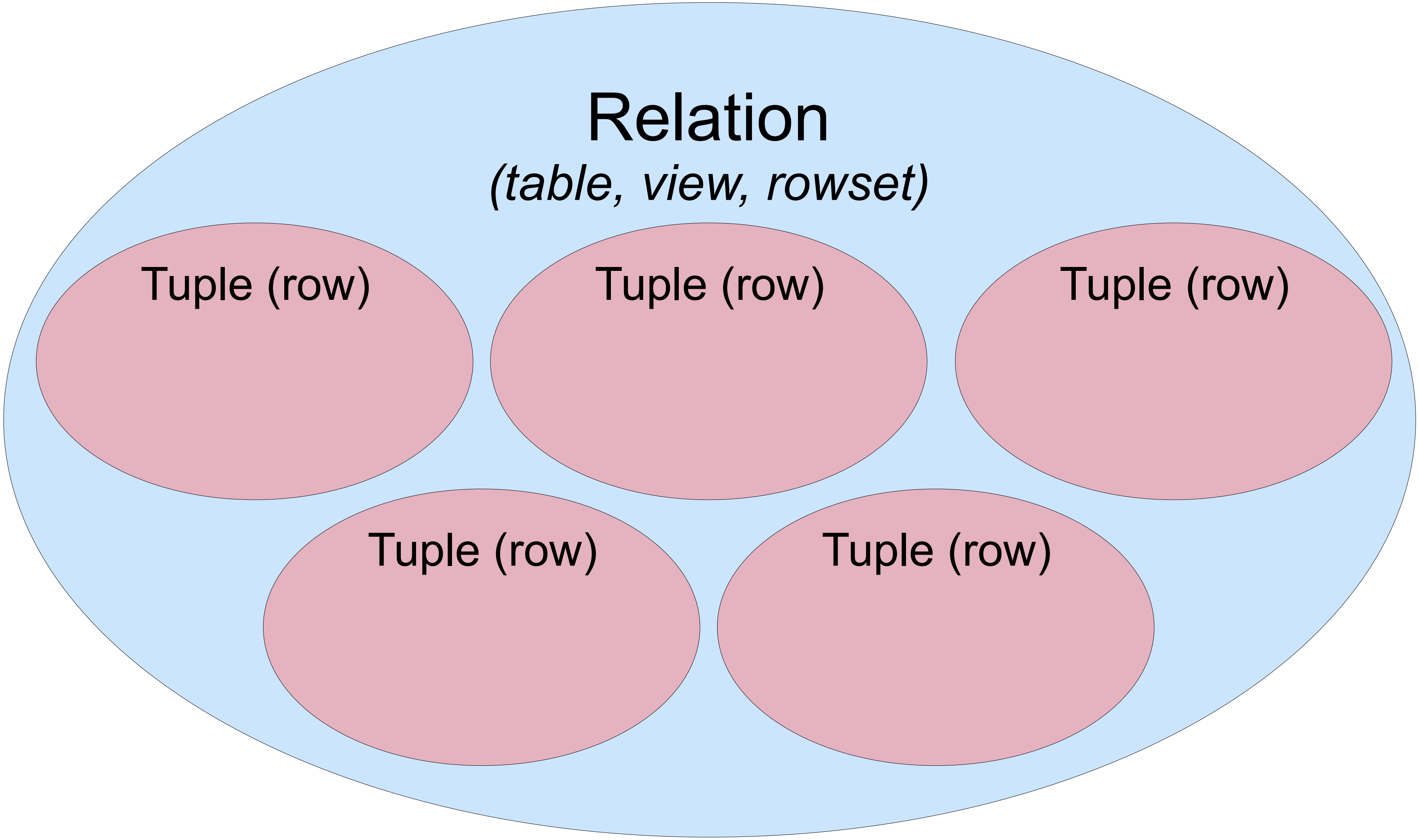
Relations



Relation

(table, view, rowset)

Tuples



Relation

(table, view, rowset)

Tuple (row)

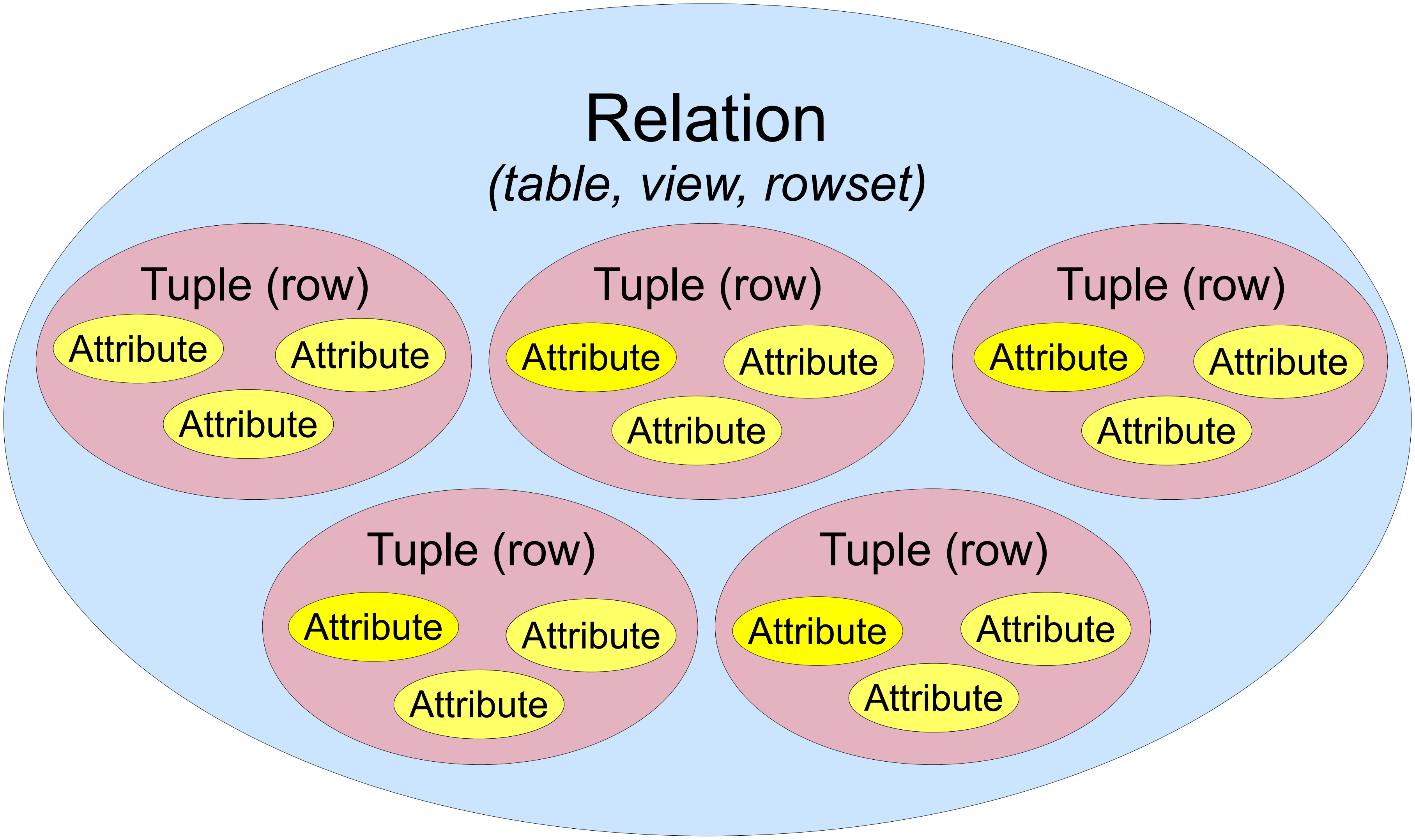
Tuple (row)

Tuple (row)

Tuple (row)

Tuple (row)

Attributes



Relation

(table, view, rowset)

Tuple (row)

Attribute

Attribute

Attribute

Tuple (row)

Attribute

Attribute

Attribute

Tuple (row)

Attribute

Attribute

Attribute

Tuple (row)

Attribute

Attribute

Attribute

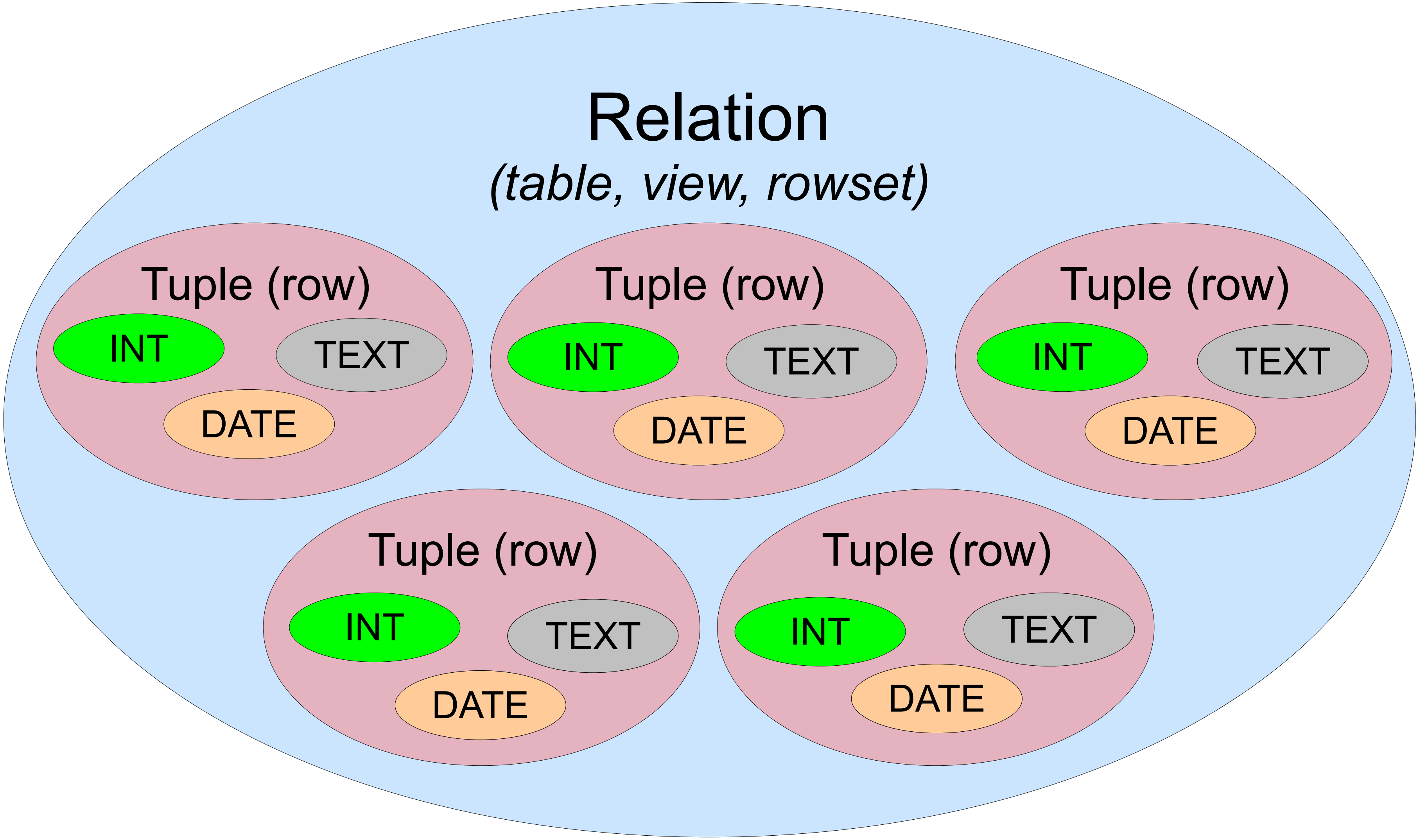
Tuple (row)

Attribute

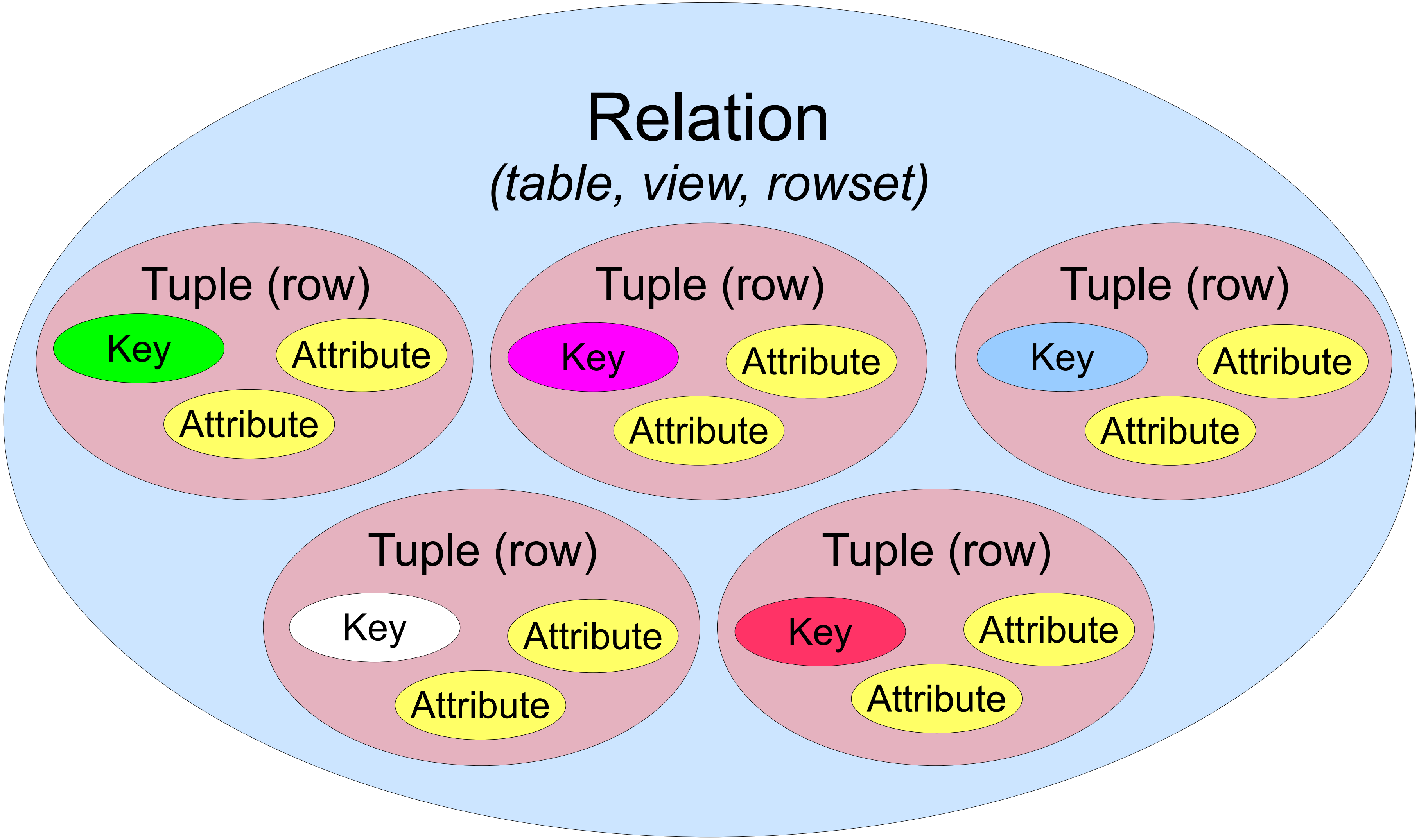
Attribute

Attribute

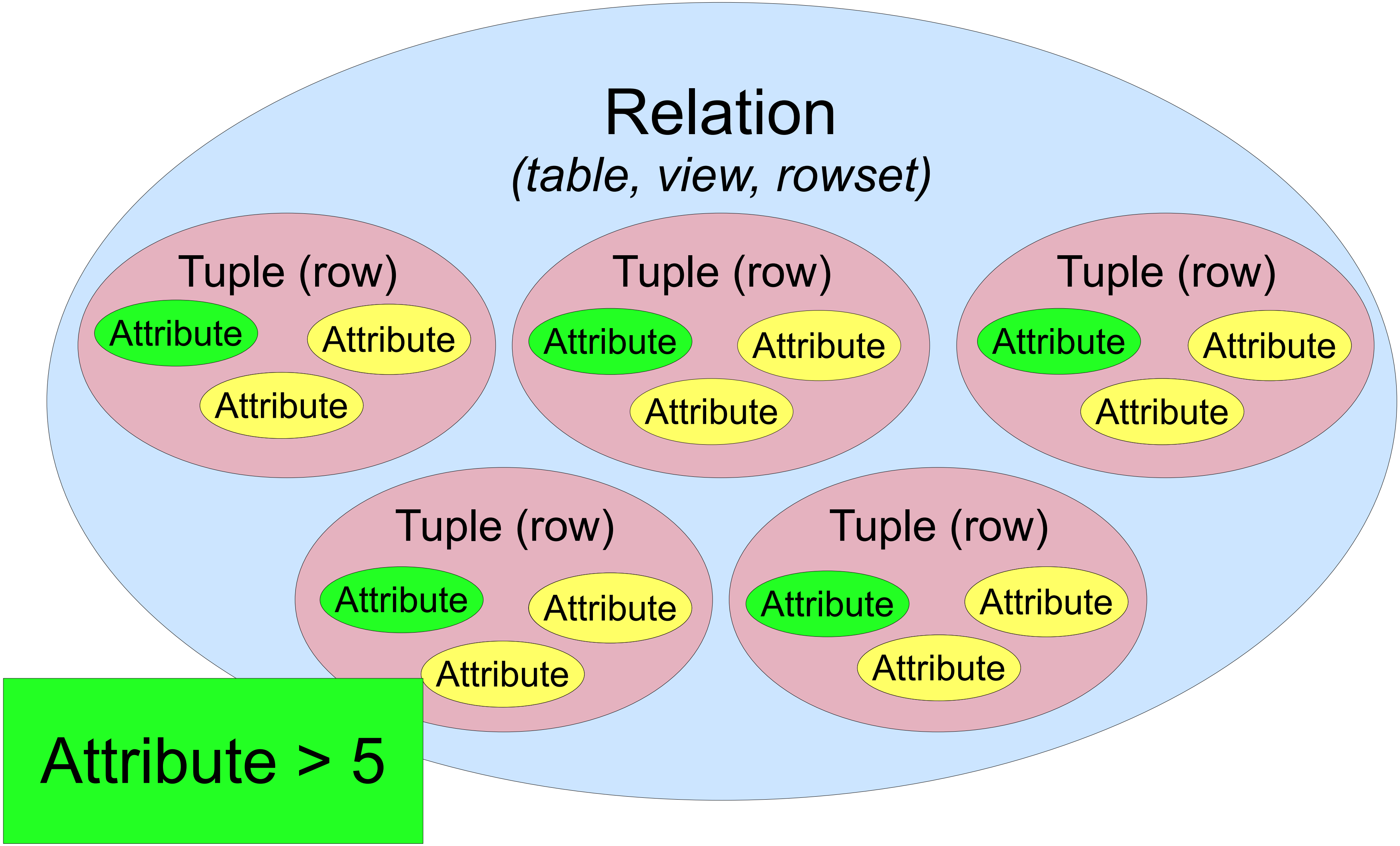
Domains (types)



Keys

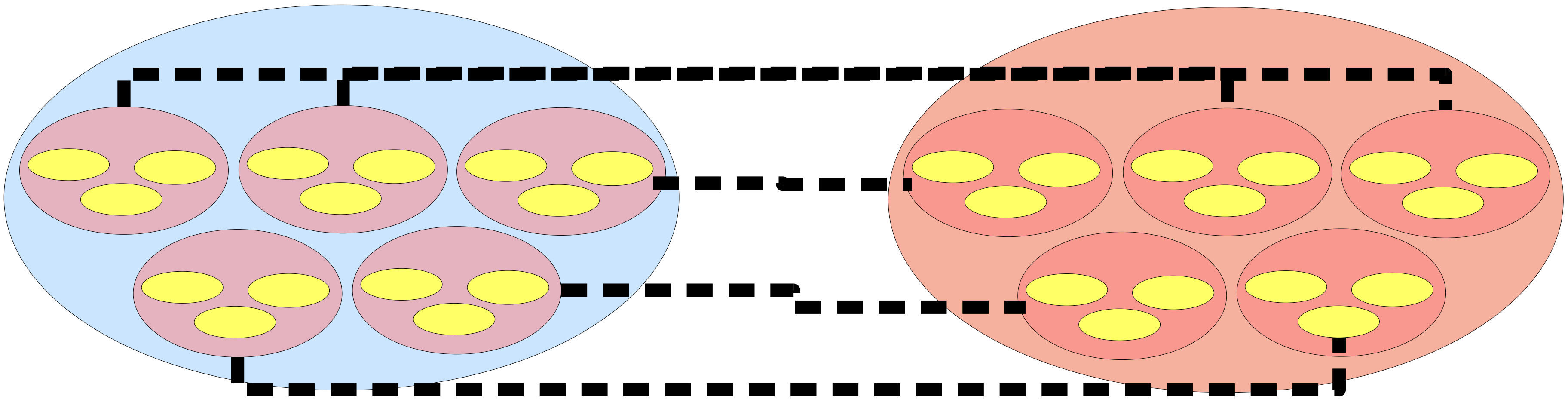

Relation
(table, view, rowset)
Tuple (row)
Key
Attribute
Attribute
Tuple (row)
Key
Attribute
Attribute
Tuple (row)
Key
Attribute
Attribute
Tuple (row)
Key
Attribute
Attribute
Tuple (row)
Key
Attribute
Attribute

Constraints

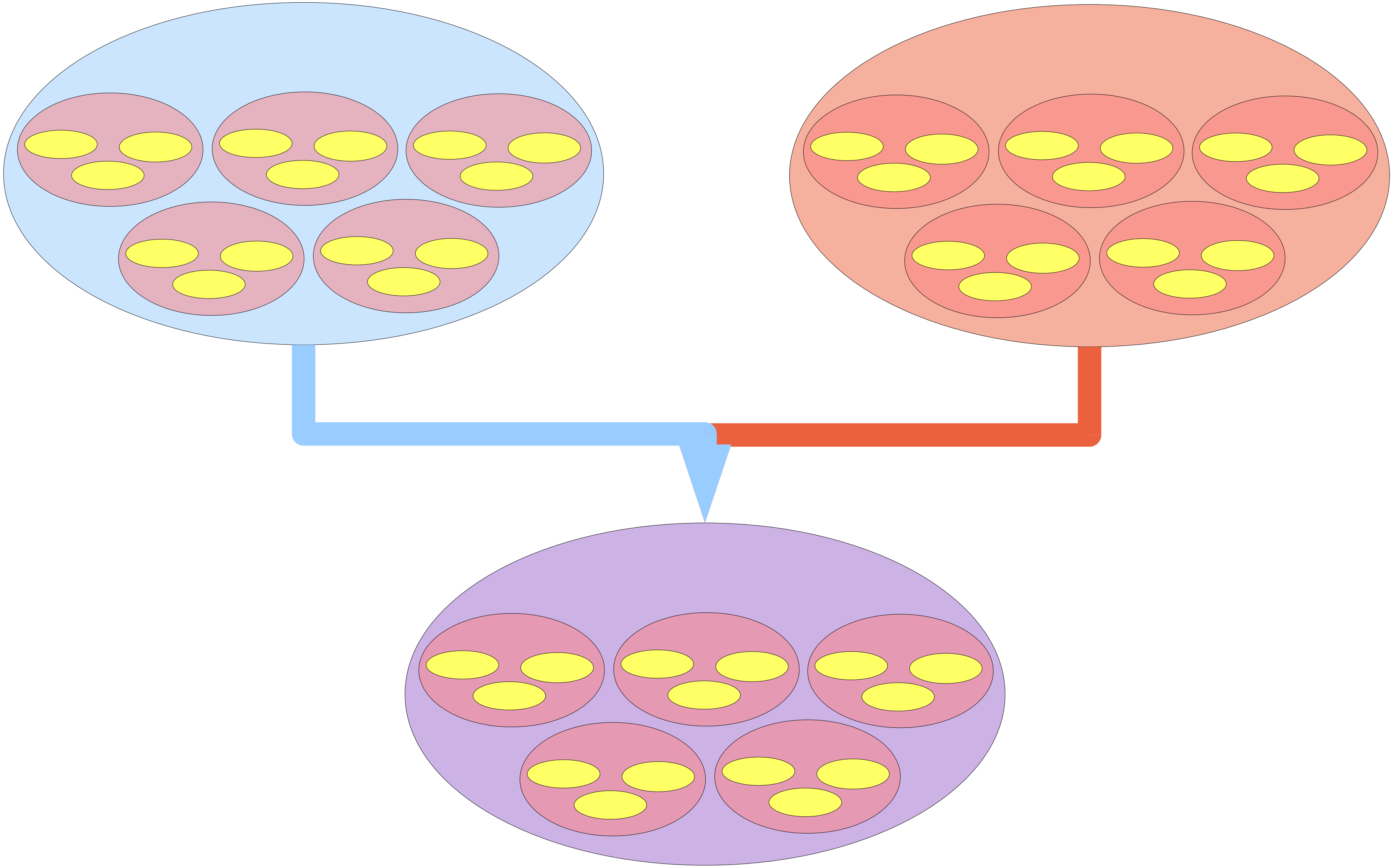


Attribute > 5

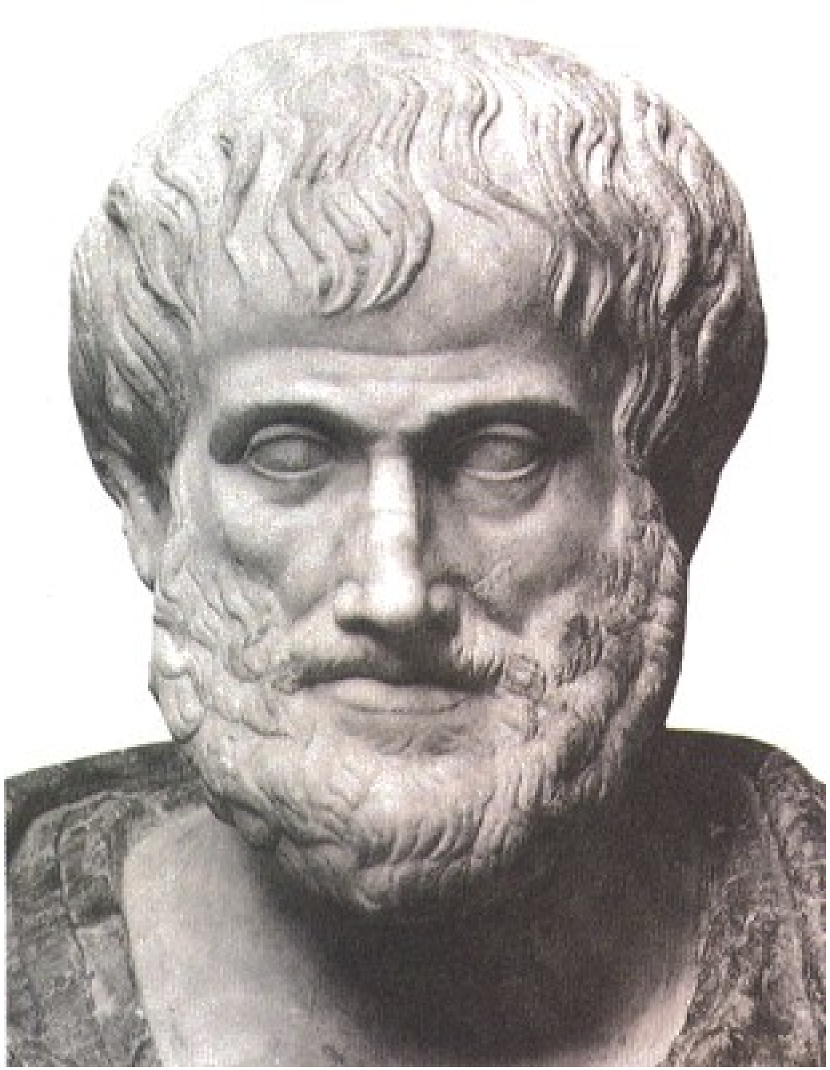
Foreign Key Constraint



Derived Relation (query)

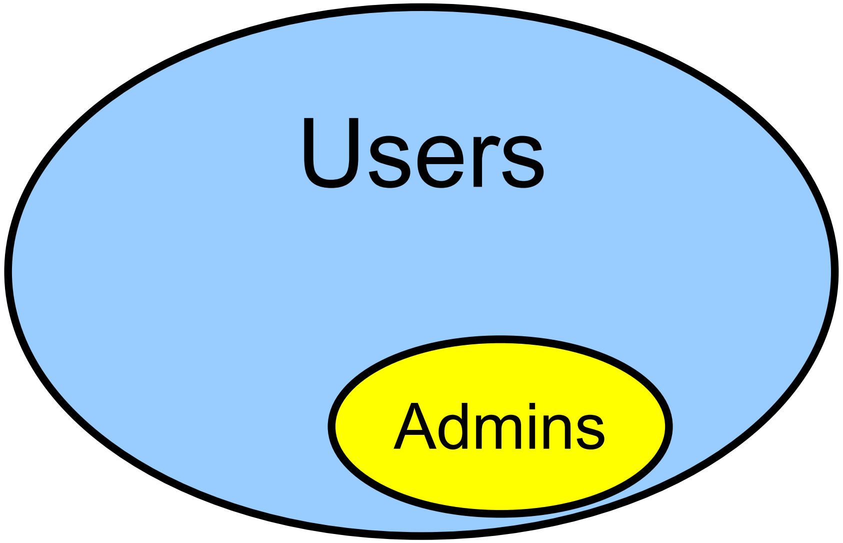


Atomic Data



Non-Atomic Attributes

- **name (text)**
- email (text)
- login (text)
- password (text)
- **status (char)**



Atomic, Shmomic. Who Cares?

- Atomic Values:
  - make joins easier
  - make constraints easier
- Non-atomic Values:
  - increase CPU usage
  - make you more likely to forget something

What's Atomic?

The simplest form of a datum, which is not divisible without loss of information.

name
Josh Berkus

```
SELECT SUBSTR(name,STRPOS(name, ' ')) ...
```

Status
a

```
… WHERE status = 'a' or status = 'u' ...
```

What's Atomic?

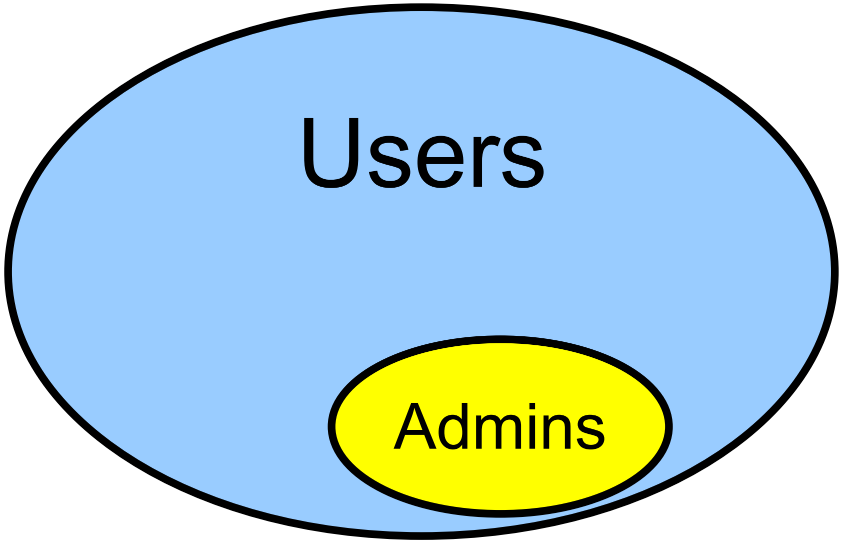
The simplest form of a datum, which is not divisible without loss of information.

first_name	last_name
Josh	Berkus

active	access
TRUE	a

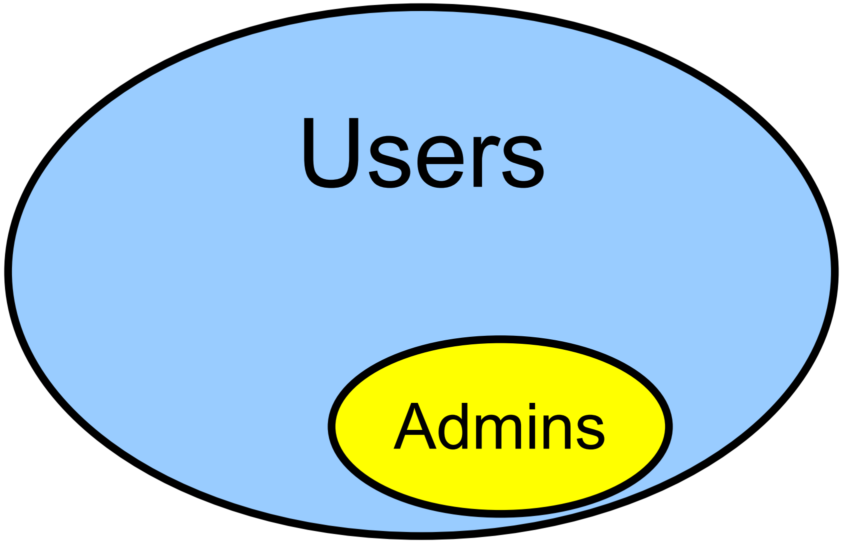
Table Atomized!

- first_name (text)
- last_name (text)
- email (text)
- login (text)
- password (text)
- active (boolean)
- access (char)



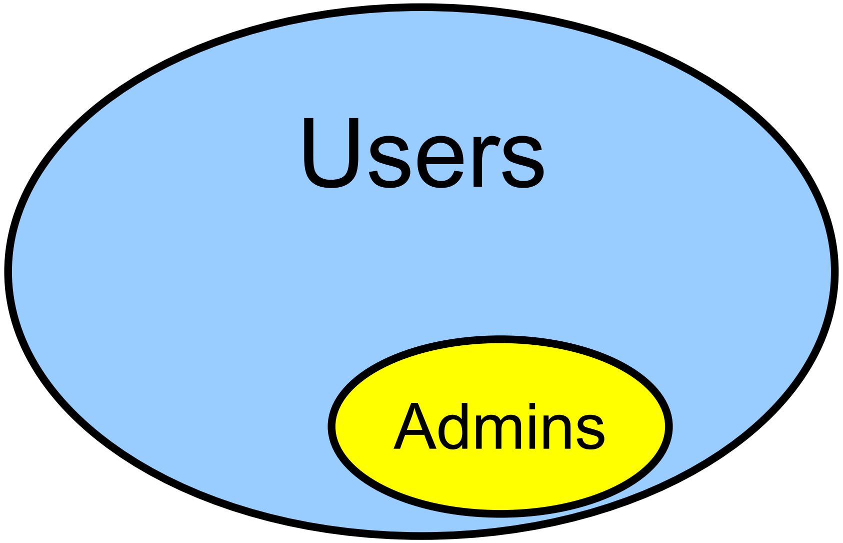
Where Are My Keys?

- first_name (text)
- last_name (text)
- email (text)
- login (text)
- password (text)
- active (boolean)
- access (char)



Candidate (Natural) Keys

- first_name (text)
- last_name (text)
- email (text)
- login (text) Key
- password (text)
- active (boolean)
- access (char)



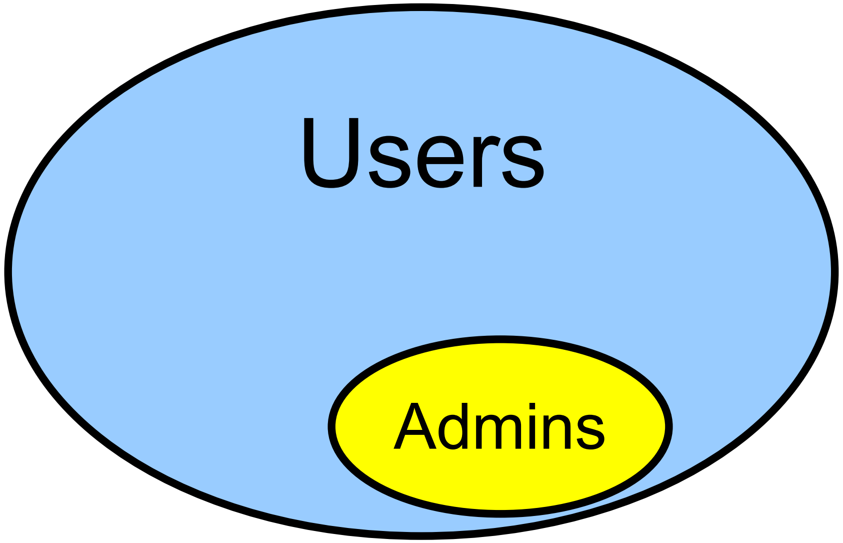
A Good Key

- Should *have* to be unique because the application requires it to be.
- Expresses a unique predicate which describes the tuple (row):
    - user with login “jberkus”
    - post from “jberkus” on “2009-05-02 13:41:22” in thread “Making your own wine”
- If you can't find a good key, your table design is missing data.

Surrogate Key

- first_name (text)
- last_name (text)
- email (text)
- login (text) Key
- password (text)
- active (boolean)
- access (char)

- user (serial)



When shouldn't I use surrogate keys?

- As a substitute for real keys
  - not *ever*
- If the real key works for the application
  - it's a single column
  - it's small
- For Join Tables (more later)
- If they are not going to be used
  - leaf tables

When *should* I use surrogate keys?

- If the real key is complex or really large
  - 4 columns
  - large text field
  - time range
- If your application framework requires them
  - but probably better to get a better framework
- If you're doing data warehousing
  - where the bytes count

But wait, aren't ID fields “faster”?

No.

While INTs are smaller,
joins are expensive.

Test twice, design once.

users: no surrogate key

```
create table users (
   first_name  text not null check
      ( length(first_name) between 1 and 40 ),
   last_name   text not null check
      ( length(last_name) between 2 and 30 ),
   login    text not null unique check
      ( length(login) between 4 and 30 ),
   password    text not null check
      ( length(login) between 6 and 30 ),
   email       email not null unique,
   description text,
   icon     text,
   level    integer not null default 1
           references access_levels (level)
           on update cascade on delete set default,
   active      boolean not null default TRUE
);
```

posts: surrogate keys

```
create table posts (
   post SERIAL not null unique,
   thread  integer not null references threads(thread)
      on delete cascade on update cascade,
   created    timestamp with time zone
              not null default current_timestamp,
   owner   text not null
        references users (login) on update cascade
        on delete cascade,
   content    text not null,
   flag    char(1) references flags(flag)
        on update cascade on delete set null
   constraint posts_key unique (thread, created, owner)
);
```

Constraints

for clean data

- Are there to prevent “bad data”.
  - allow you to rely on specific assertions being true
  - prevent garbage rows
  - deter application errors
    - and stupid display problems

Is VARCHAR(#) a Constraint?

- No, not really
  - if you need an upper limit, you probably need a lower limit
- but … data types are primitive constraints
  - just not constraining enough to prevent bad input

Defaults
for convenience

- Allow you to forget about some columns
  - help support “NOT NULL” constraints
- Let you set values for “invisible” columns
  - like auditing information
- Let you set things “automatically”
  - like created on current_timestamp

But my Application Code Takes Care of Data Format!

- Maybe
  - you probably don't want to make column constraints *too* restrictive
  - allow some room for cosmetic changes
    - and non-essential data
- Maybe Not
  - applications have bugs
  - everything has a RESTful interface now
  - NULLs can behave very oddly in queries

No Constraints

first_name	last_name	email	login	password	active	level
Josh	Berkus	josh@pgexperts.com	jberkus	jehosaphat	TRUE	a
NULL	NULL	kelley@ucb	k	NULL	FALSE	u
Mike	Hunt	www.pornking.com	c34521	c34521	TRUE	l
S	F	gavin@sf.gov	gavin	twitter	NULL	x

Constraints and Defaults

- first_name text
  - not null check (length between 1 and 40)
- last_name text
  - not null check (length between 2 and 40)
- email text not null ???
- login text
  - not null unique check (length between 4 and 40)
- password text
  - not null unique check (length between 6 and 30)

Constraints and Defaults

- active boolean
  - not null default TRUE
- access char(1)
  - not null check in('a','u') default 'u'
- user_id serial
  - not null unique

Gee, that was easy!

is that all there is?

Well, no. It gets more complicated.
See you after the break.

We All Just Want to Be Normal

Abby Normal

login	level	last_name
jberkus	u	Berkus
selena	a	Deckelman

login	title	posted	level
jberkus	Dinner?	09:28	u
selena	Dinner?	09:37	u
jberkus	Dinner?	09:44	a

Abby Normal

login	level	last_name
jberkus	u	Berkus
selena	a	Deckelman

login	title	posted	level
jberkus	Dinner?	09:28	u
selena	Dinner?	09:37	u
jberkus	Dinner?	09:44	a

How can I be “Normal”?

1. Each piece of data only appears in one relation
   - except as a “foreign key” attribute

- No “repeated” attributes

login	level	last_name
jberkus	u	Berkus
selena	a	Deckelman

login	title	posted
jberkus	Dinner?	09:28
selena	Dinner?	09:37
jberkus	Dinner?	09:44

But What's Really “Non-Repeated”?

login	level	privileges
jberkus	u	read,post,search
selena	a	read,post,search,edit,delete,ban
webcrawler	r	read
mike	u	read,post,search
carol	u	read,post,search

obviously repeated

But What's Really "Non-Repeated"?

login	level
jberkus	u
selena	a
webcrawler	r
mike	u

level	read	post	search	edit	delete	ban
u	t	t	t	f	f	f
a	t	t	t	t	t	t
r	t	f	f	f	f	f

non-repeated

But What's Really “Non-Repeated”?

login	level
jberkus	u
selena	a
webcrawler	r
mike	u
carol	u

level	name
u	user
a	administrator
r	read-only

level	privilege
u	read
u	post
u	search
r	read
a	delete
a	ban
a	post

non-repeated

How do you decide between one/several tables?

- Simple Rule: “one thought, one table”
  - like “one thought, one paragraph”
- You probably need more tables if:
  - there's no unique key
  - there's more than one unique key
- You may need less tables if:
  - you're doing lots of one-to-one joins

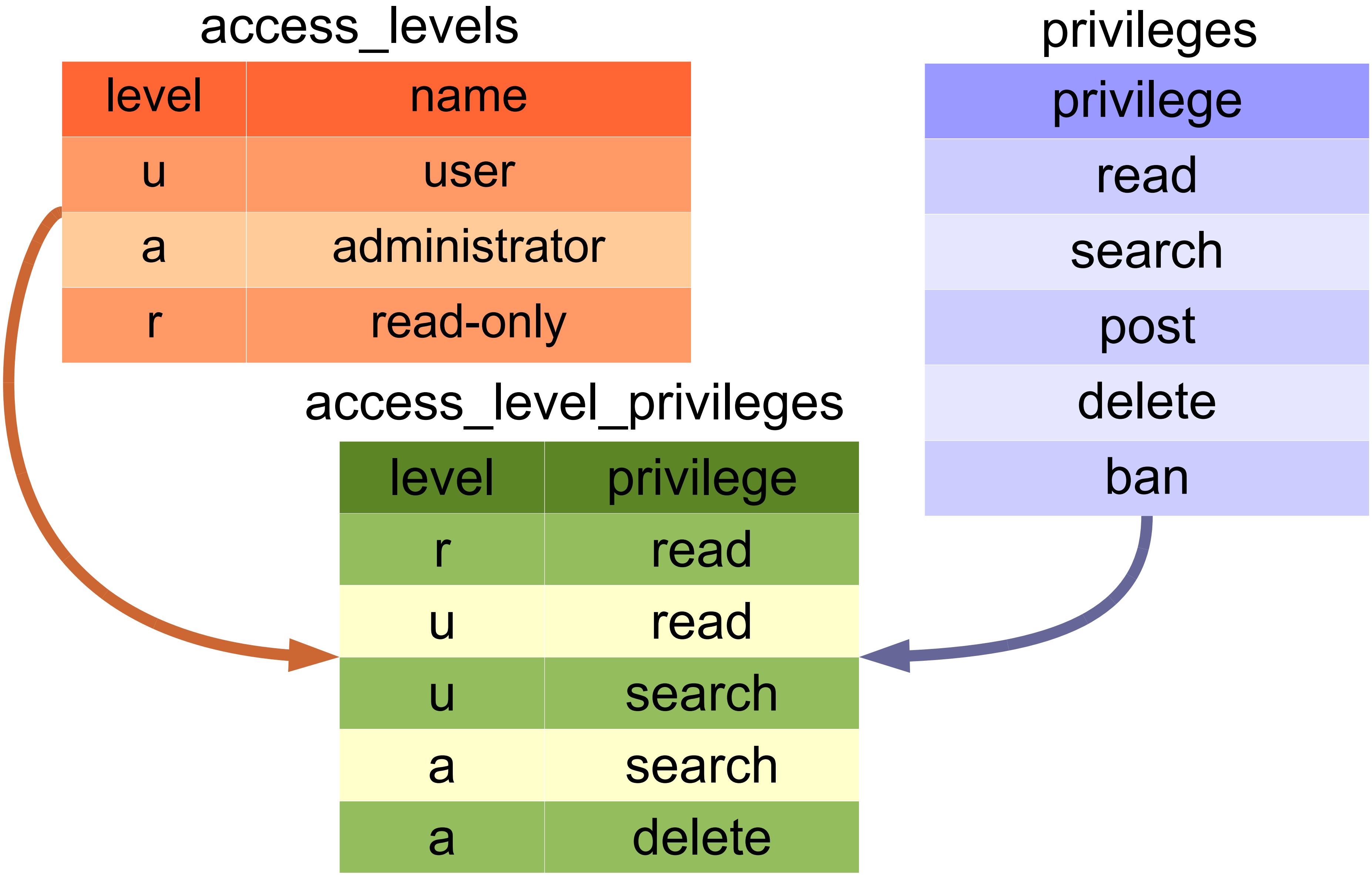
How do you decide between one/several tables?

- Otherwise, it's based on the *Application*
- how does the application use the data?
  - does it want an array?
  - use a flat series of columns
- does it want a single fact or check?
  - do you expect to add new types a lot?
  - use a vertical child table

But wait, doesn't Normalization have something to do with ID fields and everything in a lookup table?

No.

Special Case #1: many-to-many relationships



access_levels

level	name
u	user
a	administrator
r	read-only

privileges

privilege
read
search
post
delete
ban

access_level_privileges

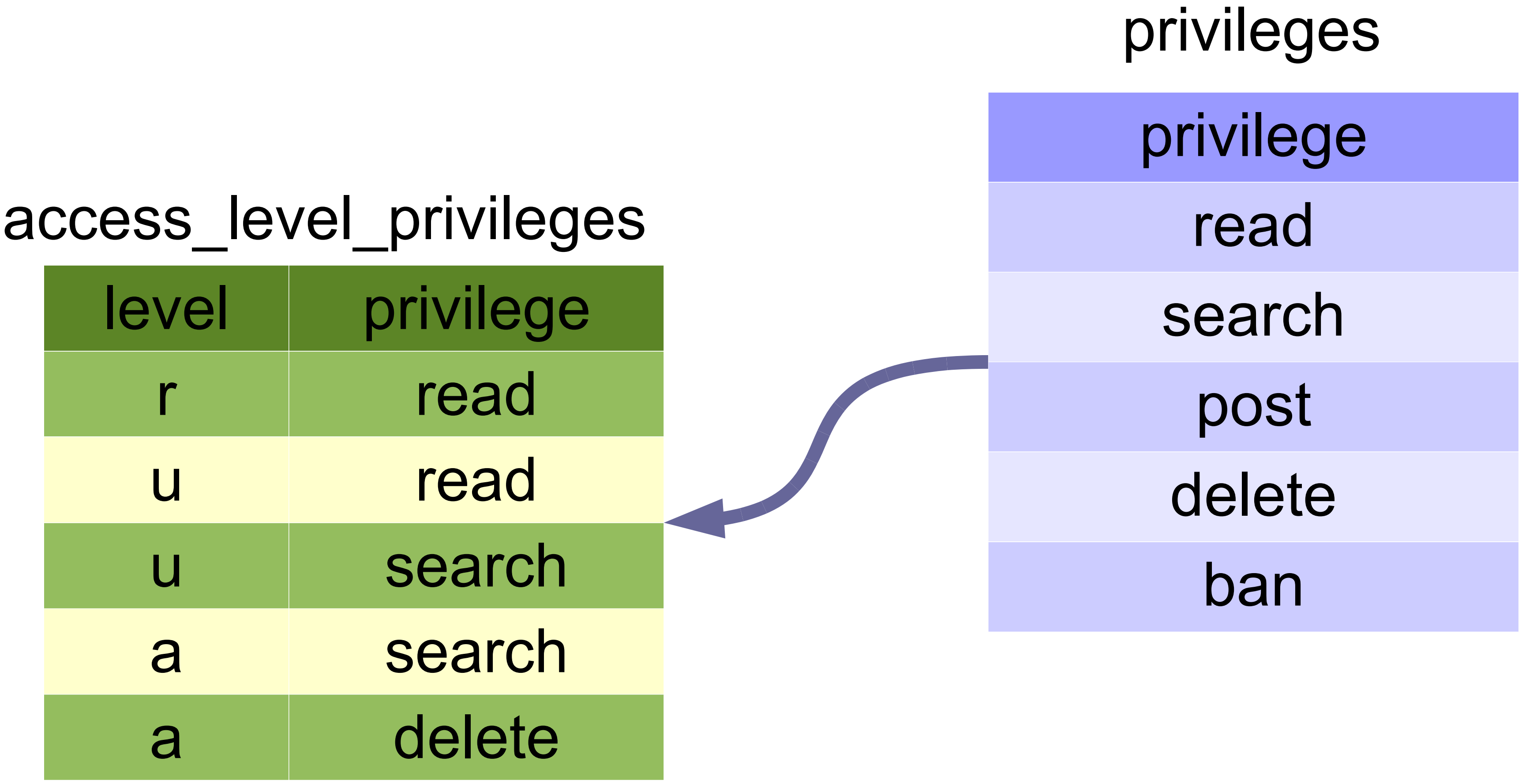
level	privilege
r	read
u	read
u	search
a	search
a	delete

Join Tables

- Contain only the keys of two or more other tables
- Should have a single unique index across all keys
- Should have Foreign Keys to all the other tables with CASCADE

Special Case #2: Lookup Tables

for constraints



privileges

privilege
read
search
post
delete
ban

access_level_privileges

level	privilege
r	read
u	read
u	search
a	search
a	delete

Special Case #2:
Lookup Tables

dimension tables

first_name	last_name	city
Josh	Berkus	2
David	Fetter	3
Selena	Deckelman	17
Miho	Ishakura	2
David	Gould	3
Robert	Treat	42
Bruce	Momjian	91

city	name	state
2	San Francisco	CA
3	Oakland	CA
17	Portland	OR
42	Washington	DC
91	Philadelphia	PA

When do I use Dimension Tables?

- When there's multiple facts/levels to the dimension
  - locations
  - demography
- When you need to save space
  - really, really big tables (millions of rows)
- Do not use them “just because”.
  - dimension tables are *not* normalization

Special Case #3: Tree Structures

- Developers want posts to “nest”
  - posts should form a tree, one under the other

- “Palio Restaurant” July 19th
  - “Re: Palio Restaurant” July 21st
    - “Re: Re: Palio Restaurant” July 23rd
    - “Re: Re: Palio Restaurant” July 24th
  - “Re: Palio Restaurant” July 23rd

Tree Structures: Proximity Tree

- Each item has a link to its parent item
  - post 34 | parent_post 21
- Advantages
  - most common
  - fast to update
- Disadvantages
  - slow to query
  - requires WITH RECURSIVE or CONNECT_BY()

Tree Structures: Path Fields

- Each item has a full “path” of its parentage
  - post 34 | path 7,21,26
- Advantages
  - fast to sort
  - fast to query & search
- Disadvantages
  - slow to update
  - requires non-standard SQL extensions
    - or text parsing

posts Table

```
create table posts (
   post  SERIAL not null unique,
   thread  integer not null references threads(thread)
              on delete cascade on update cascade,
   parent_post   integer references posts(post)
        on delete cascade on update cascade,
   created    timestamp with time zone
              not null default current_timestamp,
   owner   text not null references users (login)
           on update cascade on delete cascade,
   content    text not null,
   flag    char(1) references flags(flag)
           on update cascade on delete set null
   constraint posts_key unique (thread, created, owner)
);
```

Special Case #4: Extensible Data

- Developers want admins to be able to create “flexible profiles”
  - series of items
  - undefined at installation time
- Josh Berkus
  - male
  - bearded
  - wears glasses

Extensible Data:
Entity-Attribute-Value

ID	Property	Setting
407	Eyes	Brown
407	Height	73in
407	Married?	TRUE
408	Married?	FALSE
408	Smoker	FALSE
408	Age	37
409	Height	66in

property	format
Eyes	text
Height	number
Married?	boolean
Age	number
Smoker	boolean

EAVil

- Space-consumptive
  - many many rows, lots of row overhead
- Enforcing constraints by procedural code
  - very CPU-intensive
- Can't make anything “required”
- Can't index effectively
- Many-Way Joins
  - selecting combinations performs horribly
- however, you *can* cascade-drop

EAVil

- All unmarried men with red hair under 30

```
SELECT first_name, last_name
FROM users
   JOIN user_profiles married USING (login)
   JOIN user_profiles men USING (login)
   JOIN user_profiles hair USING (login)
   JOIN user_profiles age USING (login)
WHERE married.property = 'Married?'
        and married.value::BOOLEAN = FALSE
   AND men.property = 'Gender' and men.value = 'm'
   AND hair.property = 'Hair' and hair.value = 'Red'
   AND  age.property = 'Age' and age.value::INT < 30
```

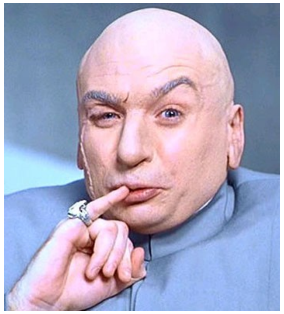
E-Blob

ID	Properties
407	<eyes=”brown”><height=”73”> <married=”1”><smoker=”1”>
408	<hair=”brown”><age=”49”> <married=”0”><smoker=”0”>
409	<age=”37”><height=”66”> <hat=”old”><teeth=”gold”>

E-Blobby

- Slow to update, slow to search
  - need to use application code or lots of parsing
- Requires special database extensions
  - XML, hstore, etc.
- Advantages over EAV
  - smaller storage space (with compression)
  - no horrible joins
  - combinations easier
  - feeds directly into application code

How to Decide: EAVil vs. ThE-Blob



- Will you be searching for specific items?
  - EAVil
- Will you be just spitting out all data to the application?
  - E-Blob
- Do you have special DB extenstions?
  - E-Blob

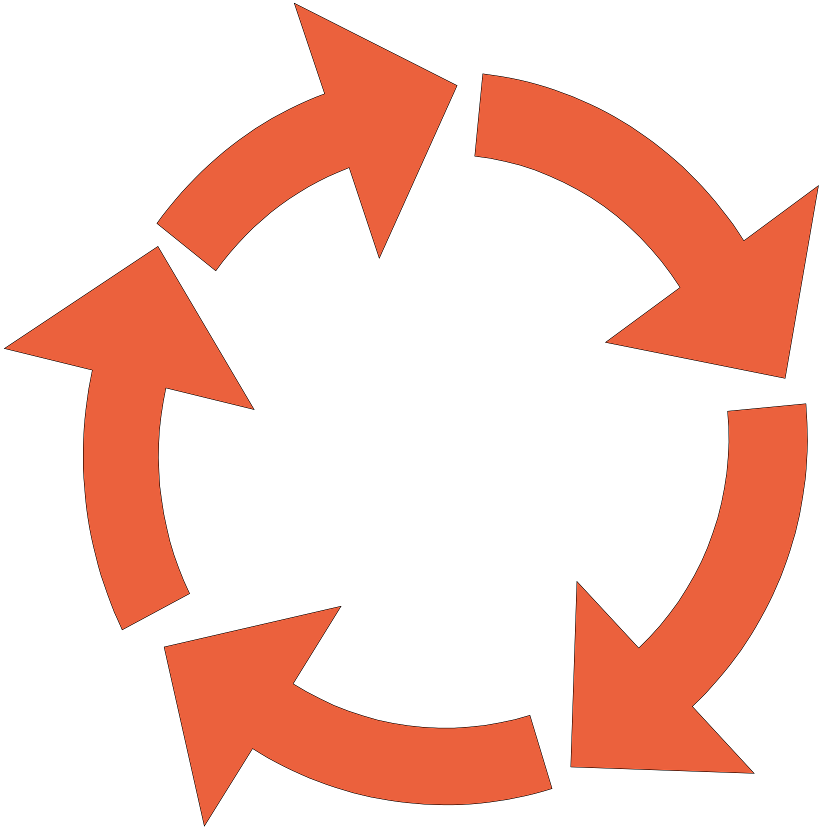
When Not to use EAV & E-Blob

- As the foundation for all of your data
  - non-relational databases do this better
- For data which has important checks and constraints
  - or is required
- For data which needs to be searched fast
- As a way of modifying your application
  - alter the database!

E-blob: The users Table

```
create table users (
   first_name  text not null
      check ( length(first_name) between 1 and 40 ),
   last_name   text not null
      check ( length(last_name) between 2 and 30 ),
   login    text not null unique
      check ( length(login) between 4 and 30 ),
   password    text not null
      check ( length(login) between 6 and 30 ),
   email       email not null unique,
   description text,
   icon     text,
   level    integer not null default 1
        references access_levels (level)
        on update cascade on delete set default,
   active      boolean not null default TRUE,
   profile     xml
);
```

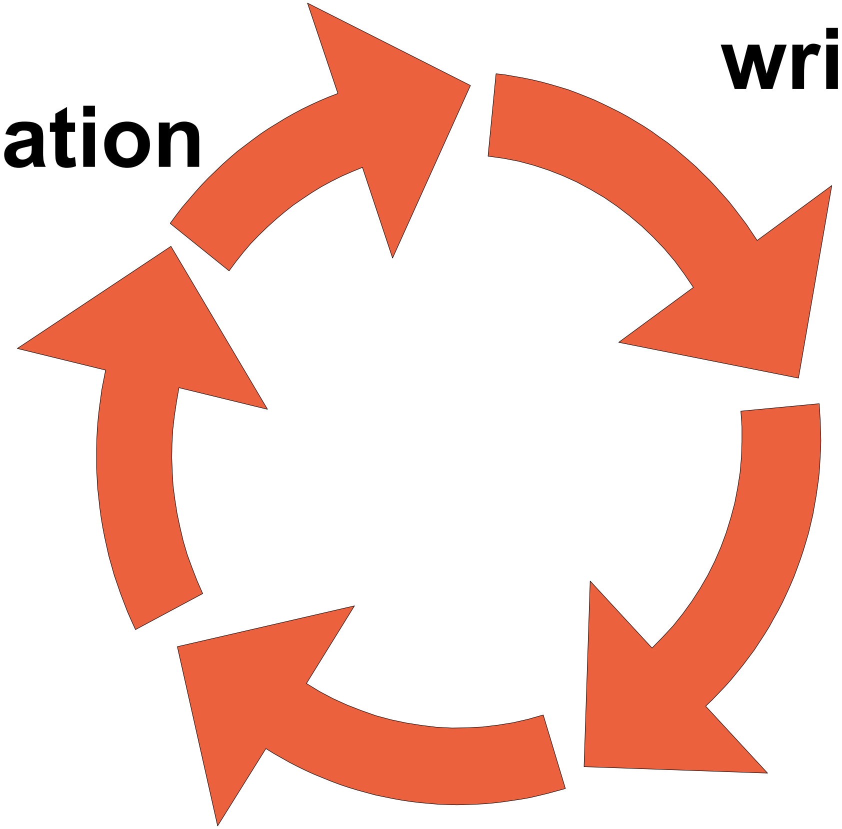
Managing Change



Making a DB Schema is a Process
not an end result

- Waterfall is Dead
  - don't make the schema static and the application dynamic
  - if you use Aglie/TDD/etc. for app, use it for DB
  - Plan to Iterate

Software Development Cycle (TDD)



create specification

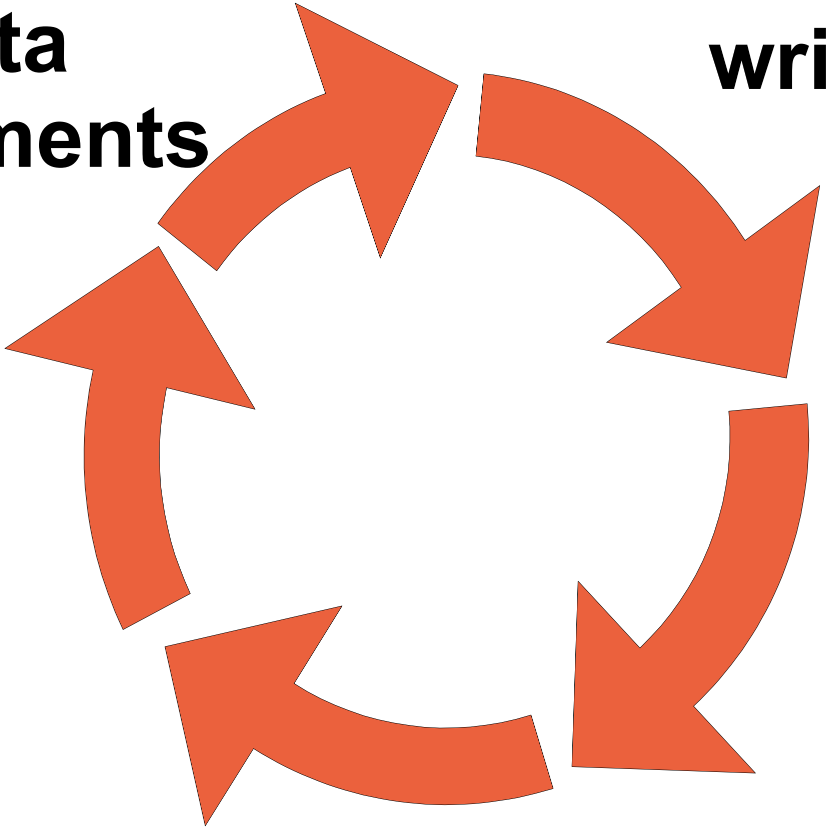
write tests

develop software

deploy new version

get user feedback

Database Development Cycle (TDD)



write data requirements

write tests

develop new schema

deploy new schema

get developer feedback

But wait, how do I manage change without breaking the application?

- The same as for software development
  1) Testing
  2) Migrations
  3) Backwards-compatible APIs

Testing

- Unit tests for database objects
  - especially stored procedures
- Application tests for application queries
  - need to be able to run all application queries and test for breakage
- Performance Regression tests
  - make sure you're not breaking performance

Migrations

- For each schema change, write a SQL migration
  - use transactional DDL (if available)
- Sequence these updates
  - tie them to application updates
- Watch out for irreversability
  - unlike application migrations, database reversions may destroy data

Backwards-Compatible API

Views

Views: Messages Table

- messages are sent from one user to one user

```
create table messages (
   message SERIAL not null unique,
   sender  text not null references users(login)
      on delete cascade on update cascade,
   recipient text not null references users(login)
      on delete cascade on update cascade,
   sent  timestamp with time zone
      not null default current_timestamp,
   subject text not null
      check (length(subject) between 3 and 200 ),
   content text not null
);
```

- developers want multiple recipients
  - but, they don't want to refactor all code

1. Create message_recipients

```
create table message_recipients (
   message int not null references
      messages(message)
      on delete cascade on update cascade,
   recipient text not null references users(login)
      on delete cascade on update cascade,
   constraint message_recipients_key
      unique ( message, recipient )
);
```

2. Rename and modify messages

```
INSERT INTO message_recipients
SELECT message, recipient FROM messages;

ALTER TABLE messages
NAME to message_contents;

ALTER TABLE message_contents
DROP COLUMN recipient;
```

3. Create VIEW for backwards compatibility

```
CREATE VIEW messages AS
SELECT message, sender,
  array_agg(recipient),
  sent, subject, content
FROM message_contents JOIN
  message_recipients
  USING ( message )
GROUP BY message, sender,
  sent, subject, content;
```

Some Good Practices

“practice doesn't make perfect,
perfect practice makes perfect.”

Consistent, Clear Naming

- Pick a Style, and Stick To It
  - plural tables or singular?
  - camel case or underscore?
  - have a “stylebook” for all developers
- Name objects what they are
  - don't abbreviate
  - don't use “cute” or “temporary” names
- If the object changes, change its name

Comment Your DB

- Use COMMENT ON … IS
  - describe each object
  - if you have time, each column
  - keep comments up to date
  - *just* like you would with application code

```
comment on table privileges is 'a list of application
privileges which can be assigned to various privilege
levels.';
```

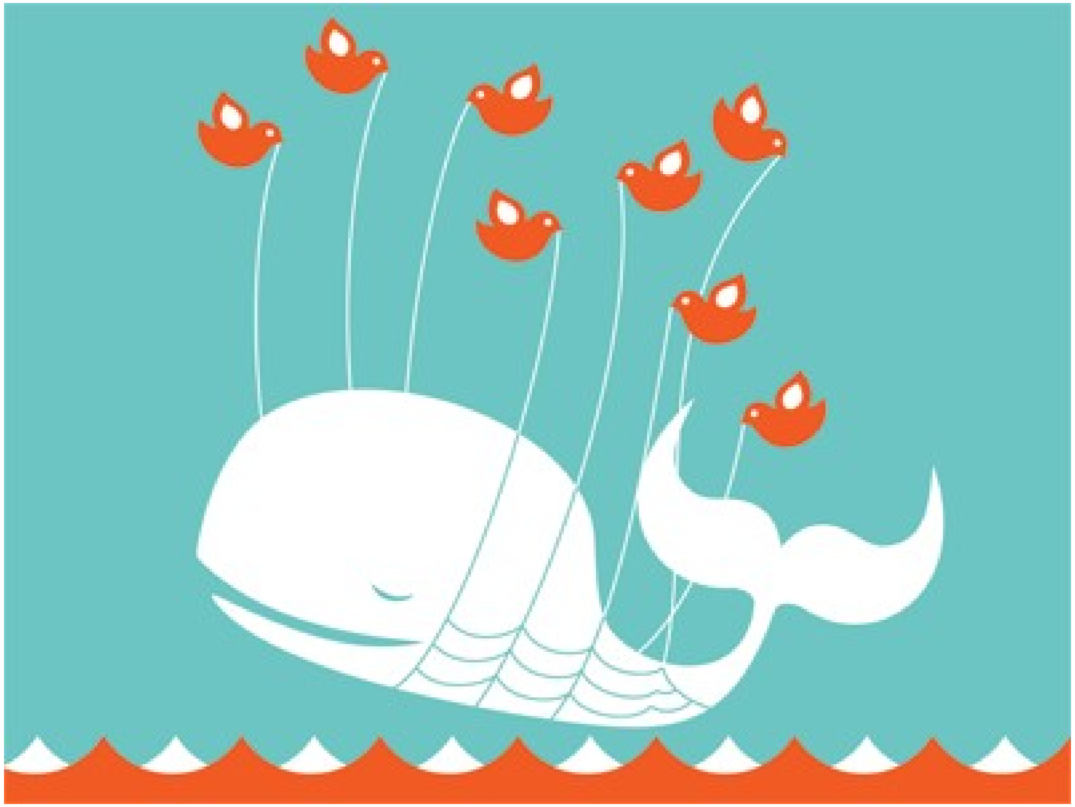
Use Source Code Management

- DDL (data definition language) is Text
  - check it into Git/SVN/Mercurial/Bazaar
  - version it

Some Bad Practices

Premature Optimization

- Don't do anything “for performance” which compromises the logical model of your data
  - unless you've tested it thoroughly first
- Poor optimization limits your throughput
  - but you can always buy more hardware
- Poor design can result in days of downtime
  - besides, database engines are designed to optimize for good design



“Downtime is more costly than slow throughput”

Premature Optimization: *Five Warning Signs*

1) Are you choosing data types according to which is “faster”?
2) Do you find yourself counting bytes?
3) Did you disable foreign keys and constraints because they were “too slow”?
4) Have you “denormalized” or “flattened” tables to make them “faster”?
5) Do you find yourself trying to override the query planner?

Polymorphic Fields

- Fields which mean different things depending on the value of another field

result	link_type	link
309	URL	http://www.postgresql.org
4718	port	443
5223	OS	88
9001	application	1915

Magic Numbers

ID = 0

2009-02-30

2000-01-01

-1, 1,2,3,4,5 100

Summary

1) The database is a *simplified* model of the problem you're solving

2) It can be designed *simply* by working with the development team on creating lists

3) Relational Theory is *simple* and has only a few rules.

4) Normalization *simply* means removing duplication

Summary

5)Designing a Table in 5 simple steps:

1)list your attributes

2)make them atomic

3)choose data types

4)choose keys

5)add constraints and defaults

Summary

6) For any given set of data, there are several possible structures: pick the one the application likes.

7) Dimension tables aren't for everyone.

8) Four Special Cases require Special SQL:

   1) Many-to-Many Join Tables

   2) Lookup tables and Dimension Tables

   3) Tree Structured Data

   4) Extensible Data

Summary

9) Managing Changes

    1) Testing

    2) Migrations

    3) Views & Procedures as Compatible API

10) Follow Good Practices

11) Avoid Bad Practices

More Information

- me
  - josh@pgexperts.com
  - www.pgexperts.com
  - it.toolbox.com/blogs/database-soup
- postgresql
  - www.postgresql.org
- at OSCON
  - PostgreSQL booth
  - State of Lightning Talks (Thursday 1:45)