Schema Design Patterns for a Peta-Scale World

Aaron Kimball

Chief Architect, WibiData

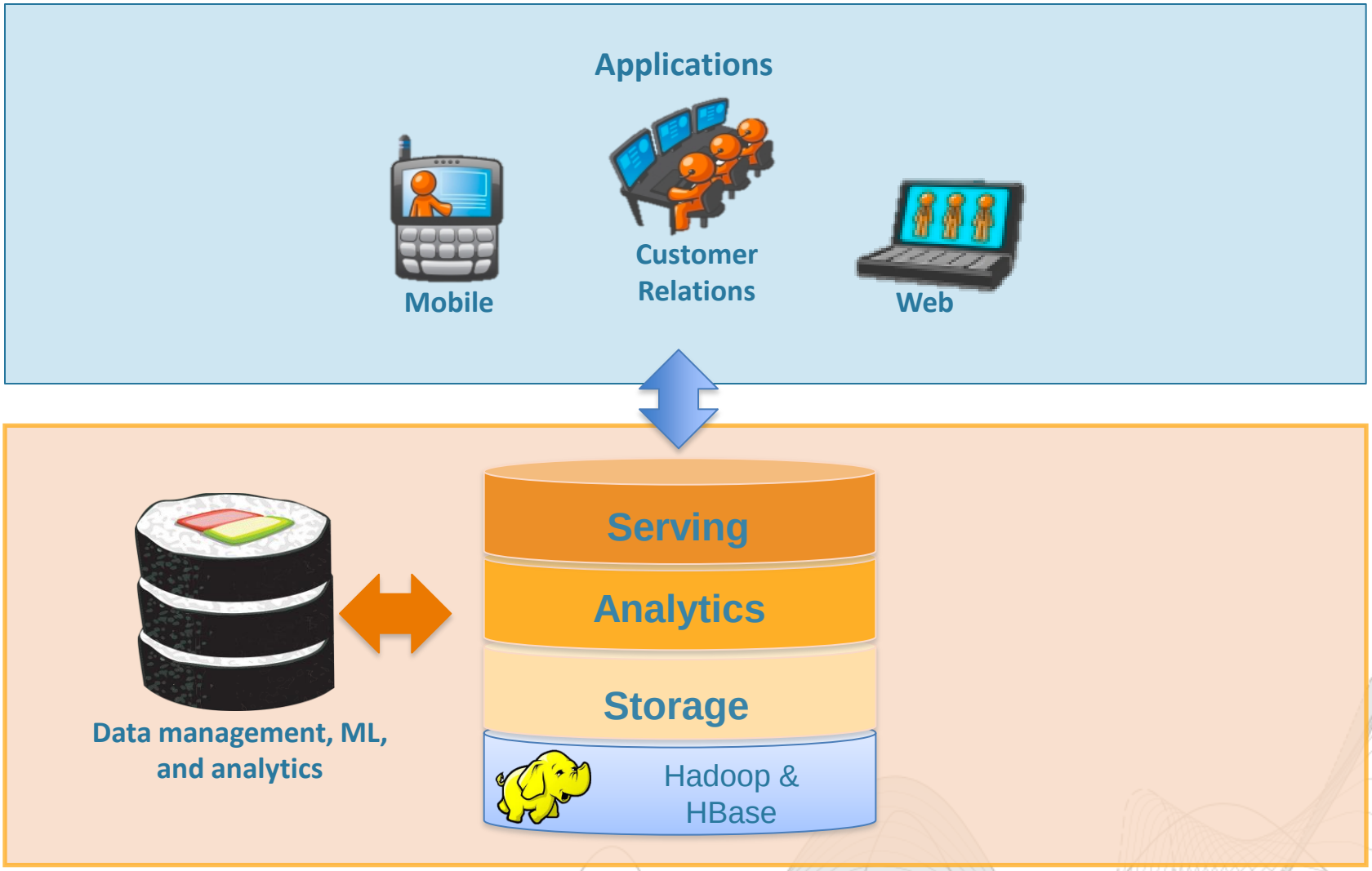
About me


UNIVERSITY OF
WASHINGTON


cloudera
Ask Bigger Questions

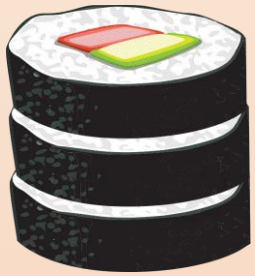

wibidata

Big Data Applications



Major application targets

Real Time Models



SQL Analytics

Batch Processing

MapReduce,
Cascading,
Crunch,
Pig…

Big Data Apps are hard to build

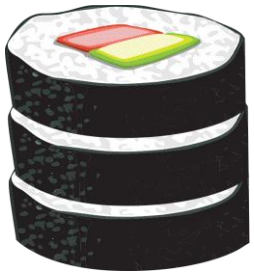
- Schemas, serialization & versioning
- Distributed deployment
- Communication between teams
- Batch + real-time capacity planning

kiji

Kiji is designed to help you build real-time Big Data Applications on Apache HBase

+

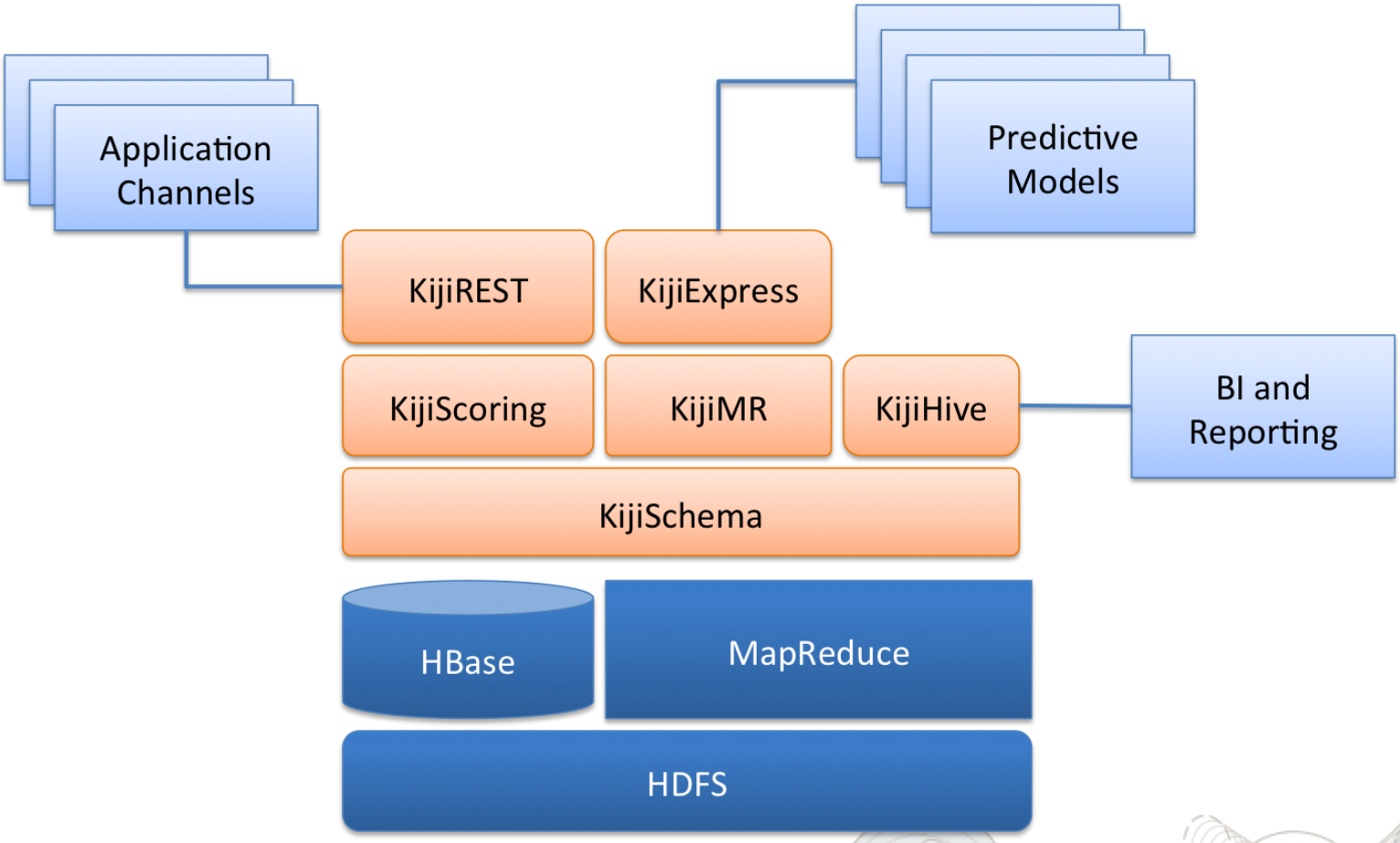
+



100% Apache 2 licensed

wibidata

Kiji architecture



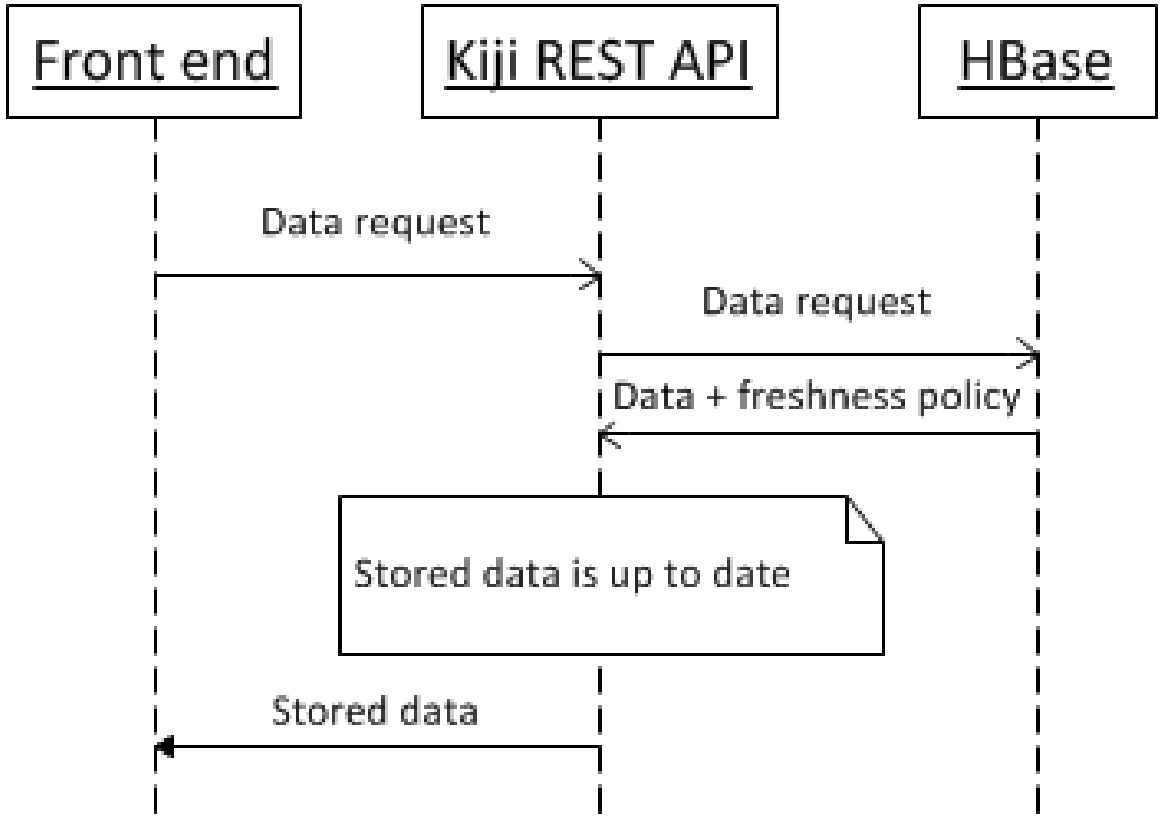
Leading design decisions

- Store your data in HBase
- Encode it using Avro
- An *entity-centric* table design
- Manage a data dictionary around tables
- Distribute writes across the cluster

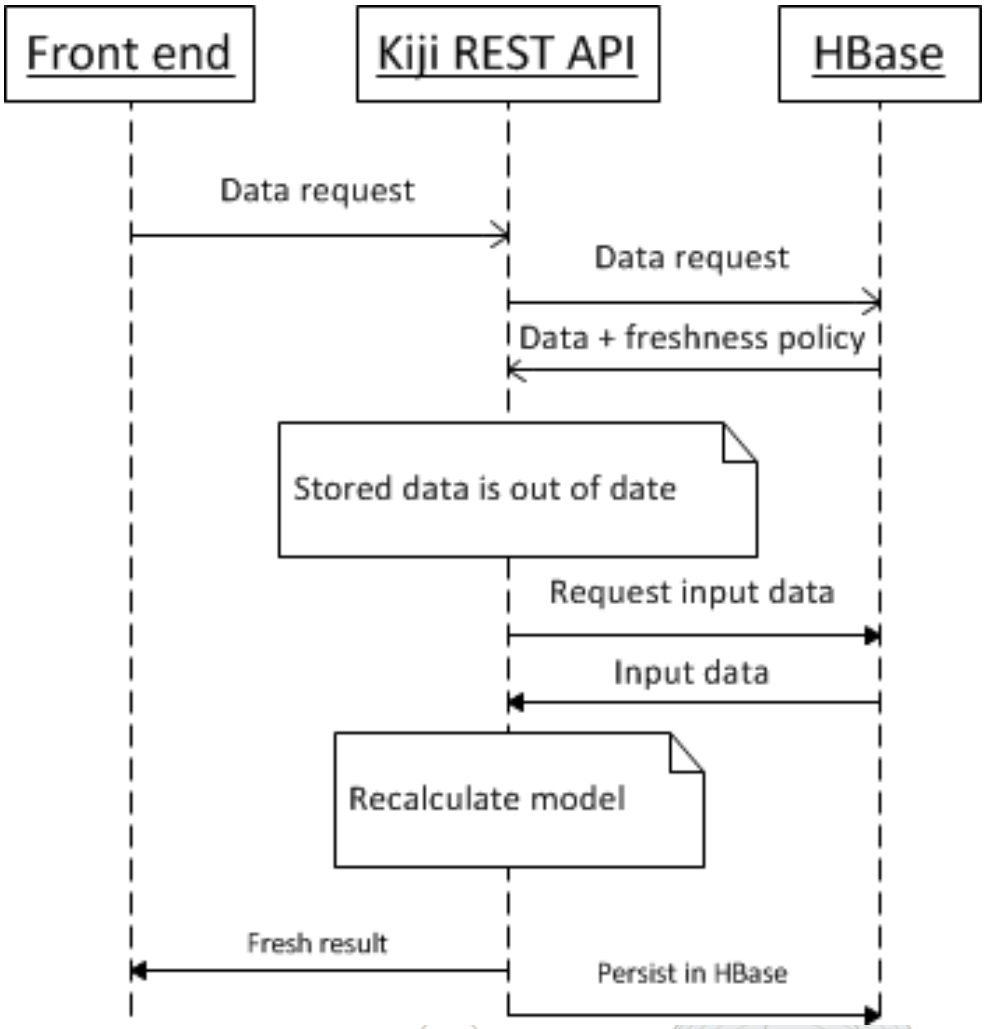
Key features

- Work with big data in rich types with schema evolution
- Guides users to successful schema design
- Deployment of real-time model scoring

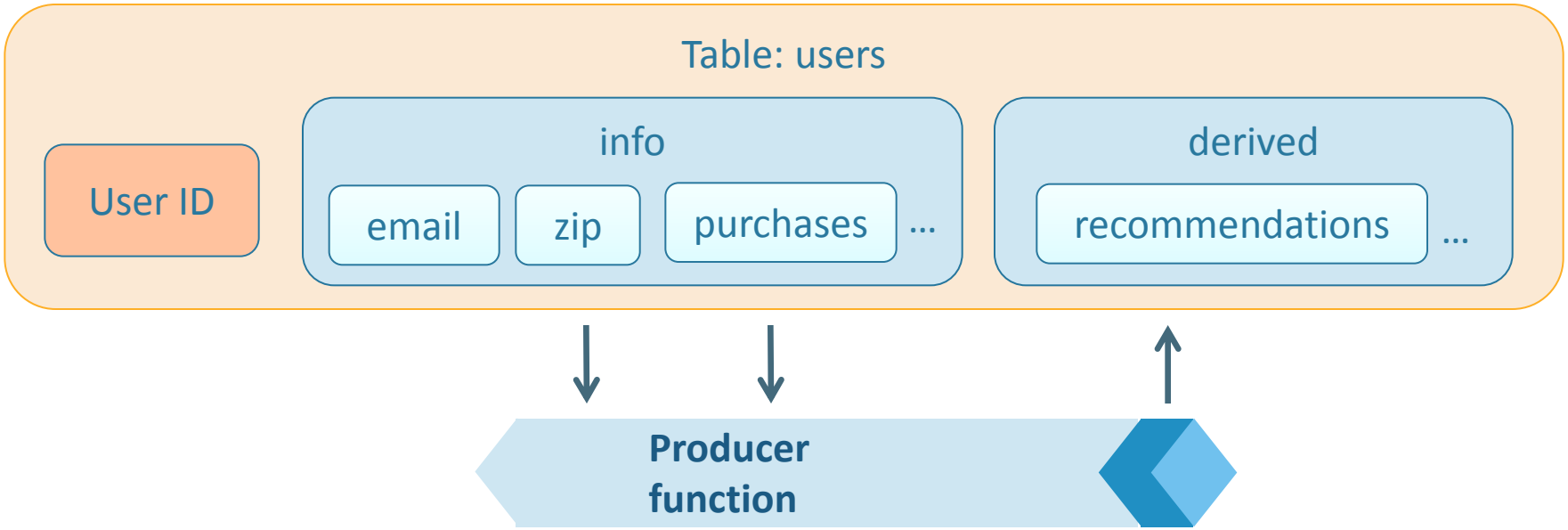
Processing big data in real time



Real time “freshening”



Producer functions

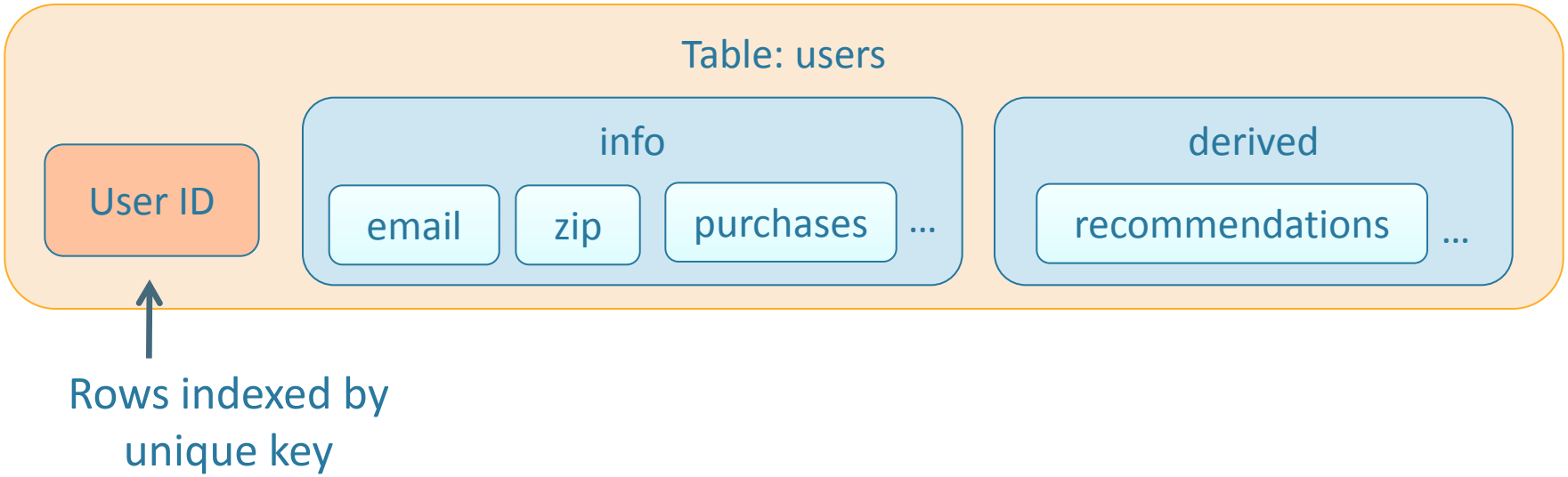


Producers provide custom per-entity analytics operating on fine-grained data

The Kiji data model

- Kiji tables have a *layout* (schema) describing their columns and data types
- Each column can hold integers or strings as well as complex Avro records

The Kiji data model

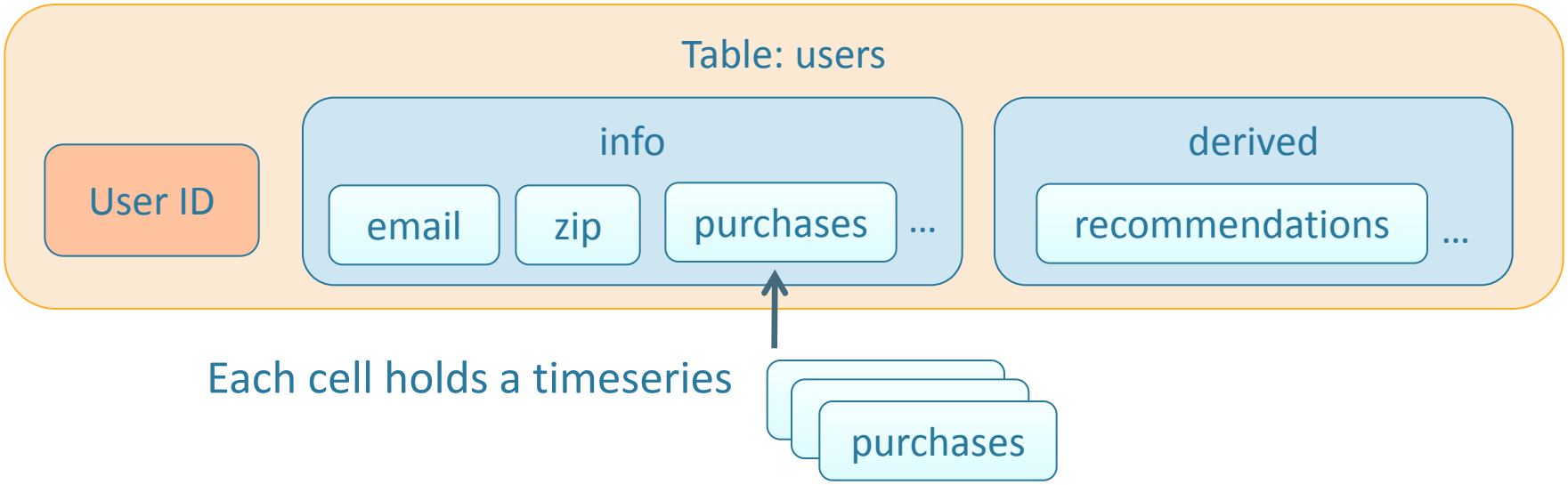


In this *entity-centric* model, all data for one entity (e.g., user) sits in one row

The Kiji data model

Data is grouped together logically in *column families*

The Kiji data model



Historical values for a given (row, column) can be easily aggregated, iterated, or filtered

Why  ?

- Schema evolution
- Compact representation
- Rich set of types
- Generic API is attractive for framework level
- Plays nicely with Hadoop ecosystem (Input/OutputFormats, etc.)

Schema evolution

Initial schema for info:location

```
record Point {
    float x;
    float y;
}
```

Later schema adds a field

```
record Point {
    float x;
    float y;
    float z = 0.0;
}
```

Applications expecting either the old or new data structure can read all records from disk, regardless of which was used when writing the data.

Efficient I/O with Avro

- Each row (entity) stores many complex-valued cells
- Data logically read and written together is physically stored in the same cell
- Sparse column storage allows fine-grained access to disparate elements of an entity for complex model building

Schema upgrade concerns

- Big data repositories are accessed by many versions of different apps concurrently
- Apps each evolve local schema definitions at different rates
- “Flag day” approaches to using new schemas do not work in practice
- Avro admits incompatible schema migrations

Incompatible migration example

- #1 `record R { int x; string y; }`
- #2 `record R { int x; }`
- #3 `record R { int x; int y = 0; }`

Incompatibility detection requires a history of all schemas previously used

Poly-schema key-value storage

- Each column records lists of all current and prior schemas
  - Separate lists for reader and writer schemas
- DDL for registering and unregistering them

```
ALTER TABLE t ADD READER SCHEMA s...
ALTER TABLE t DROP READER SCHEMA s2...
```

- Zookeeper used to check active clients

Conclusions

- Store data in an entity-centric fashion
- Hash-prefix row keys to avoid hot spots
- Use complex records in each column
  - Granularity of records is app-specific tradeoff
- Each column is *poly-schema*: several potential views at a time!

Try Kiji today!

- Go to kiji.org and download the BentoBox
  - Zero-config Hadoop + HBase + Kiji instance
  - “Batteries included”
- 15-minute quickstart guide and a tutorial with full source code

EXPERIENCE

DATA-DRIVEN

APPLICATIONS

aaron@wibidata.com

wibidata