NoSE: Schema Design for NoSQL Applications

Michael J. Mior, Kenneth Salem, Ashraf Aboulnaga, Rui Liu
• NoSQL App Development
• Problem Formulation
• NoSE Design and Implementation
• Evaluation
NoSQL

- Eventually consistent, horizontally scalable, flexible schema

- Many different types of NoSQL databases
  - Document stores
  - Key-value stores
  - Graph databases
  - ...
  - Extensible record stores
CREATE COLUMNFAMILY "ReservationsByGuest"(
    "GuestID" uuid, "ResID" uuid, 
    "ResStartDate" timestamp, 
    "RoomID" uuid, PRIMARY KEY(("GuestID"), 
    "ResStartDate", "ResID", "RoomID") )

Partitioning key

Clustering key
Database Application Development

1. Define application requirements
2. Decide on a data model for the target system
3. Implement the application according to the model
   a. Database access
   b. Application logic
Database Application Development

1. Define application requirements
2. Decide on a data model for the target system
3. Implement the application according to the model
   a. Database access
   b. Application logic
Schema Design Best Practices

Schema Design Best Practices

Model column families around query patterns

Schema Design Best Practices

Model column families around query patterns

But start your design with entities and relationships, if you can

Schema Design Best Practices

Model column families around query patterns

But start your design with entities and relationships, if you can

De-normalize and duplicate for read performance

Schema Design Best Practices

Model column families around query patterns

But start your design with entities and relationships, if you can

De-normalize and duplicate for read performance

But don’t de-normalize if you don’t need to

Schema Design Best Practices

Model column families around query patterns

But start your design with entities and relationships, if you can

De-normalize and duplicate for read performance

But don’t de-normalize if you don’t need to

Leverage wide rows for ordering, grouping, and filtering

Schema Design Best Practices

Model column families around query patterns

But start your design with entities and relationships, if you can

De-normalize and duplicate for read performance

But don’t de-normalize if you don’t need to

Leverage wide rows for ordering, grouping, and filtering

But don’t go too wide

Schema Design Example

For a given guest, return the cities that guest has stayed in

```
CREATE COLUMNFAMILY "CitiesByGuest" ("GuestID" uuid,
    "City" text, PRIMARY KEY(("GuestID"), "City"));
```

```
CREATE COLUMNFAMILY "HotelsByGuest" ("GuestID" uuid,
    "HotelID" uuid, PRIMARY KEY(("GuestID"), "HotelID"));
```

```
CREATE COLUMNFAMILY "HotelsByID" ("HotelID" uuid,
    "HotelCity" text, PRIMARY KEY(("HotelID"), "HotelCity"));
```
NoSE Overview

NoSE

- Conceptual schema
- Workload
- Selected column families
- Query implementation plans
Application Workload

For a given guest, return the cities that guest has stayed in

```
SELECT Hotel.HotelCity FROM Hotel.Room.Reservation.Guest
WHERE Guest.GuestID = ?
```
NoSE Architecture

- NoSE
  - Candidate Enumeration
  - Query Planning
  - Schema Optimization
  - Plan Recommendation

- Selected column families
- Query implementation plans

- Conceptual schema
- Workload
Query Planning Example

```
SELECT Name FROM Hotel WHERE Hotel.State = 'NY' AND Hotel.Reservation.Room.Guest.GuestID = ? ORDER BY Name
```
Schema Optimization

Construct a linear program to optimize execution time

\( C_{ij} \)  Cost of using column family \( j \) to answer query \( i \)

\( \delta_{ij} \)  Use of column family \( j \) for query \( i \) in the final plan

\( \delta_j \)  Presence of column family \( j \) in final schema

\( s_j \)  Size of column family \( j \)
Schema Optimization

Add constraints to ensure each query has a valid plan

\[
\sum_i \sum_j C_{ij} \delta_{ij} \quad \text{Minimize the cost}
\]

\[
\delta_{ij} \leq \delta_j, \forall i, j \quad \text{Ensure column families used are present}
\]

\[
\sum_j s_j \delta_j \leq S \quad \text{Limit maximum storage space}
\]
Updates

- Updates make denormalization more expensive
- Add statements to update conceptual entities
- New column families are added to support updates
- Costs for updates are added to the linear program
**Evaluation**

- Application defined by the RUBiS online auction benchmark
- Generate a schema and query plans recommended by NoSE
- Two schemas for comparison
  - Normalized (as much as possible)
  - Expert-selected
Evaluation - Schema Performance

![Bar chart showing average response time for different transaction types across NoSE, Normalized, and Expert categories.](chart)

- **Transaction type**:
  - BrowseCategories
  - ViewUserInfo
  - AboutMe
  - RegisterUser

- **Average response time (ms)**
  - NoSE
  - Normalized
  - Expert

The chart illustrates the performance comparison across different schema performance categories for various transaction types, highlighting the response time in milliseconds.
Conclusion

- NoSE automates schema design for NoSQL applications
- Conforms to best practices without requiring expertise
- Schemas are better than those produced manually with an average of 1.8x and up to 125x performance improvement
Questions?

git.io/nose-icde