Polyglot database architectures = polyglot challenges

Lena Wiese

Institute of Computer Science
University of Göttingen
Goldschmidtstraße 7
37077 Göttingen, Germany
lena.wiese@uni-goettingen.de

Abstract. We categorize polyglot database architectures into three types (polyglot persistence, lambda architecture and multi-model databases) and discuss their advantages and disadvantages.

1 Polyglot Database Architectures

When designing the data management layer for an application, several database requirements may be contradictory. For example, regarding access patterns some data might be accessed by write-heavy workloads while others are accessed by read-heavy workloads. Regarding the data model, some data might be of a different structure than other data; for example, in an application processing both social network data and order or billing data, the former might usually be graph-structured while the latter might be semi-structured data. Regarding the access method, a web application might want to access data via a REST interface while another application might prefer data access with query language. It is hence worthwhile to consider a database and storage architecture that includes all these requirements. We describe three option for polyglot database architectures in the following three sections.

1.1 Polyglot Persistence

Instead of choosing just one single database management system to store the entire data, so-called **polyglot persistence** could be a viable option to satisfy all requirements towards a modern data management infrastructure. Polyglot persistence (a term coined in [4]) denotes that one can choose as many databases as needed so that all requirements are satisfied. Polyglot persistence can in particular be an optimal solution when backward-compatibility with a **legacy application** must be ensured. The new database system can run alongside the legacy

 In: R. Bergmann, S. Görg, G. Müller (Eds.): Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB. Trier, Germany, 7.-9. October 2015, published at http://ceur-ws.org

database system; while the legacy application still remains fully functional, novel requirements can be taken into account by using the new database system.

An implementation of a data processing system that connects to several data sources and integrates and merges data from these sources is Apache Drill [2]. Apache Drill is inspired by the ideas developed in Google's Dremel system [6].

It should obviously be avoided to push the burden of all of these query handling and database synchronization task to the application level – that is, in the end to the programmers that maintain the data processing applications. Instead it is usually better to introduce an integration layer. The integration layer then takes care of processing the queries – decomposing queries in to several subqueries, redirecting queries to the appropriate databases and recombining the results obtained from the accessed databases; ideally, the integration layer should offer several access methods, and should be able to parse all the different query languages of the underlying database systems as well as potentially translate queries into other query languages. Moreover, the integration layer should ensure **cross-database consistency**: it must synchronize data in the different databases by propagating additions, modifications or deletions among them.

Polyglot persistence however comes with severe disadvantages:

- Uniform access: There is no unique query interface or query language, and hence access to the database systems is not unified and requires knowledge of all needed database access methods.
- Consistency: Cross-database consistency is a major challenge because referential integrity must be ensured across databases (for example if a record in one database references a record in another database) and in case data are duplicated (and hence occur in different representation in several databases at the same time) the duplicates have to be updated or deleted in unison.
- Interoperability: The underlying database systems are developed independently. Newer versions of databases may not be interoperable with the integration layer and the administrator has to keep track of frequent updates.
- Logical Redundancy: Logical redundancy can only be avoided with a database design that strictly assigns non-intersecting subsets of the data to different databases. This might contradict some access requirements of users.
- Security: Access control must be enforced by the integration layer and all connected databases have to be configured to only allow restricted access.

1.2 Lambda Architecture

When real-time (stream) data processing is a requirement, a combination of a slower batch processing layer and a speedier stream processing layer might be appropriate. This architecture has been recently termed **lambda architecture** [5]. The lambda architecture processes a continuous flow of data in the following three layers:

Speed Layer: The speed layer collects only the most recent data. As soon as data have been included in the other two layers (batch layer and serving layer), the data can be discarded from the speed layer dataset. The speed layer incrementally computes some results over its dataset and delivers these results in several **real-time views**; that is, the speed layer is able to adapt his output based on the constantly changing data set. Due to the relatively small size of the speed layer data set, the runtime penalty of incremental computations are still within acceptable limits.

Batch Layer: The batch layer stores all data in an append-only and immutable fashion in a so-called master dataset. It evaluates functions over the entire dataset; the results are delivered in so-called **batch views**. Computing the batch views is an inherently slow process. Hence, recent data will only be gradually reflected in the results.

Serving Layer: The serving layer makes batch views accessible to user queries. This can for example be achieved by maintaining indexes over the batch views.

User queries can be answered by merging data from both the appropriate batch views and the appropriate real-time views.

An open source implementation following the ideas of a lambda architecture is Apache Druid [3] that processes streaming data in real-time nodes and batch data in historical nodes.

In practice, the lambda architecture often relies on external storage ("deep storage" in case of Druid) or stream processors (on the input side). Due to this it only has slight advantages over the polyglot persistence approach. Moreover this architecture is mostly geared towards real-time processing of data and less to ad-hoc querying.

1.3 Multi-Model Databases

Relying on different storage backends increases the overall complexity of the system and raises concerns like inter-database consistency, inter-database transactions and interoperability as well as version compatibility and security. It might hence be advantageous to use a database system that stores data in a single store but provides access to the data with different APIs according to different data models. Databases offering this feature have been termed **multi-model** databases. Multi-model databases either support different data models directly inside the database engine or they offer layers for additional data models on top of a single-model engine.

Two open source multi-model databases are OrientDB [7] and ArangoDB [1]. OrientDB offers a document API, an object API, and a graph API; it offers extensions of the SQL standard to interact will all three APIs. Alternatively, Java APIs are available. The Java Graph API is compliant with Tinkerpop [8]. ArangoDB is a multi-model database with a graph API, a key-value API and a document API. Its query language AQL (ArangoDB query language) resembles SQL in parts but adds several database-specific extensions to it.

Several advantages come along with this single-database multi-model approach:

– Reduced database administration: maintaining a single database installation is easier than maintaining several different database installations in parallel, keeping up with their newest versions and ensure inter-database compatibility. Configuration and fine-tuning database settings can be geared towards a single database system.
– Reduced user administration: In a multi-model database only one level of user management (including authentication and authorization) is necessary.
– Integrated low-level components: Low-level database components (like memory buffer management) can be shared between the different data models in a multi-model database. In contrast, polyglot persistence with several database systems requires each database engine to have its own low-level components.
– Improved consistency: With a single database engine, consistency (including synchronization and conflict resolution in a distributed system) is a lot easier to ensure than consistency across several different database platforms.
– Reliability and fault tolerance: Backup just has to be set up for a single database and upon recovery only a single database has to be brought up to date. Intra-database fault handling (like hinted handoff) is less complex than implementing fault handling across different databases.
– Scalability: Data partitioning (in particular "auto-sharding") as well as profiting from data locality can best be configured in a single database system – as opposed to more complex partitioning design when data are stored in different distributed database systems.
– Easier application development: Programming efforts regarding database administration, data models and query languages can focus on a single database system. Connections (and optimizations like connection pooling) have to be managed only for a single database installation.

2 Conclusion

Data come in different formats and data models. Modern data stores support advanced data management in the native data models [9]. Polyglot database architectures can handle several different data models at a time.

Polyglot persistence can respond to differing user demands; however it comes at the cost of increased administration overhead and more complex configuration (in particular in terms of security). Hence, polyglot persistence can only be recommended if several diverse data models have to be supported and the maintenance overhead can be managed.

The lambda architecture is a good choice for real-time data analytics but also relies on external data storage with similar disadvantages as polyglot persistence.

Multi-model databases are a good choice if only a limited set of data models is required by the accessing applications. Multi-model excel in terms of administration effort and security and hence are optimal, when only the limited set of data formats supported by the multi-model database are needed.

References

1. ArangoDB: Https://www.arangodb.com/
2. Drill: Http://drill.apache.org/
3. Druid: Http://druid.io/
4. Fowler, M.J., Sadalage, P.J.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Prentice Hall (2012)
5. Marz, N., Warren, J.: Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co. (2015)
6. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment 3(1-2), 330–339 (2010)
7. OrientDB: Http://orientdb.com/
8. Tinkerpop: Http://tinkerpop.incubator.apache.org/
9. Wiese, L.: Advanced Data Management – for SQL, NoSQL, Cloud and Distributed Databases. DeGruyter/Oldenbourg (2015)