The Shared-Data Architectural Style

A Comprehensive Reference for Data-Centric Distributed Systems

Contents

1 Overview

The shared-data style is a component-and-connector architectural style that structures a system around one or more data repositories accessed by multiple data accessor components. Communication between components is mediated by the shared data store rather than through direct component-to-component interaction. This data-centric approach places persistent data at the heart of the architecture, with components reading from and writing to the shared repository.

The fundamental characteristic of this style is indirect communication through data. Data producers write to the repository; data consumers read from it. Producers and consumers need not know about each other—they share only knowledge of the data structure and location. This indirection provides powerful decoupling, enabling components to be added, modified, or removed without affecting other components that access the same data.

The shared-data style encompasses a broad family of architectures, from traditional database-centric applications to modern data lakes and distributed data platforms. The common thread is the central role of persistent data storage in mediating component interactions and maintaining system state.

1.1 Scope and Applicability

The shared-data style applies to systems where persistent data is central to the architecture. This includes database-centric applications where business applications store and retrieve data from shared databases, data warehousing and analytics where multiple tools access consolidated data for analysis and reporting, content management systems where content is stored centrally and accessed by various presentation and management components, collaborative systems where multiple users or processes work with shared information, integration through data where systems exchange information by reading and writing shared data stores, blackboard systems where multiple knowledge sources contribute to and read from a shared problem-solving space, configuration management where shared configuration data controls system behavior, and caching architectures where shared caches reduce load on backend systems.

The style is particularly valuable when multiple components need access to the same data, when data must persist beyond the lifetime of individual components, when data producers and consumers should be decoupled, when the data model is more stable than the components that use it, and when data integrity and consistency are critical concerns.

1.2 Historical Context

The shared-data style has deep roots in computing history.

File-based systems in early computing used shared files as the primary integration mechanism between programs. Batch processing systems read input files and produced output files consumed by subsequent programs.

Database management systems emerged to provide structured, concurrent access to shared data. Relational databases with SQL became the dominant paradigm for business applications.

Data warehousing consolidated data from multiple sources for analytical processing, separating operational and analytical workloads.

Distributed databases and NoSQL systems addressed scalability and flexibility requirements beyond traditional relational systems.

Modern data platforms including data lakes, data meshes, and streaming platforms continue the evolution of shared-data architectures for big data and real-time analytics.

Understanding this evolution helps architects select appropriate data sharing patterns for their context.

1.3 Relationship to Other Styles

The shared-data style relates to several other architectural patterns.

It contrasts with direct component communication styles like client–server where components interact directly rather than through data.

It relates to publish–subscribe in that both decouple producers from consumers, but shared-data uses persistent storage rather than message passing.

It can be combined with layered architectures where the data layer provides shared storage for upper layers.

It supports pipe-and-filter when intermediate results are stored in shared repositories between processing stages.

It underlies many service-oriented architectures where services share databases or data services.

Many systems combine shared-data with other styles. A microservices system might use event-driven communication between services while each service uses shared-data internally.

1.4 Repository vs. Blackboard

The shared-data style has two major variants distinguished by control flow.

In the repository variant, data accessors initiate all actions. Components decide when to read from or write to the repository. The repository is passive, simply storing and retrieving data on request.

In the blackboard variant, the data store can trigger actions in data accessors. When data changes, the blackboard notifies relevant components. Components may be activated based on the current state of shared data. This is common in AI and expert systems where knowledge sources respond to evolving problem state.

Both variants share the fundamental characteristic of communication through shared data, but differ in where control resides.

2 Elements

The shared-data style comprises three categories of elements: repository components that store data, data accessor components that use data, and connectors that enable data reading and writing.

2.1 Repository Components

A repository component stores data persistently and provides access to that data for multiple accessor components. The repository is the central element of the style, mediating all data-related

interactions.

2.1.1 Types of Repository Components

Repository components vary in the type and structure of data they manage.

Relational databases store structured data in tables with defined schemas, supporting SQL queries and ACID transactions. Examples include PostgreSQL, MySQL, Oracle, and SQL Server.

Document databases store semi-structured data as documents (typically JSON or XML), providing flexibility in data structure. Examples include MongoDB, Couchbase, and Amazon DocumentDB.

Key-value stores provide simple storage and retrieval by key, optimized for high performance and scalability. Examples include Redis, Amazon DynamoDB, and Riak.

Wide-column stores organize data into column families, supporting sparse data and high write throughput. Examples include Apache Cassandra, HBase, and Google Bigtable.

Graph databases store nodes and relationships, optimized for traversing connected data. Examples include Neo4j, Amazon Neptune, and JanusGraph.

Time-series databases optimize for time-stamped data, supporting efficient storage and querying of temporal data. Examples include InfluxDB, TimescaleDB, and Prometheus.

Object stores provide scalable storage for unstructured data as objects with metadata. Examples include Amazon S3, Azure Blob Storage, and MinIO.

File systems provide hierarchical storage of files and directories, ranging from local file systems to distributed systems like HDFS.

Data lakes store raw data in native formats, supporting diverse analytical workloads. Examples include platforms built on S3, HDFS, or Azure Data Lake.

In-memory data grids provide distributed caching with data grid capabilities. Examples include Hazelcast, Apache Ignite, and Redis Cluster.

2.1.2 Essential Properties of Repository Components

When documenting repository components, architects should capture several property categories.

Data properties describe what is stored. Data types enumerate the types of data stored (structured, semi-structured, unstructured). Data model describes the logical organization (relational, document, graph, etc.). Schema describes the structure of stored data and how it is managed. Data volume indicates the amount of data stored and growth expectations.

Performance properties characterize repository efficiency. Read performance describes query latency and throughput. Write performance describes insert, update, and delete performance. Indexing describes how data is indexed for efficient access. Caching describes internal caching mechanisms.

Distribution properties describe data placement. Data distribution describes how data is distributed across nodes (replication, partitioning, sharding). Geographic distribution describes placement across regions or data centers. Consistency model describes the consistency guarantees provided (strong, eventual, causal).

Access properties describe who can use the repository. Number of accessors permitted indicates capacity limits on concurrent access. Access control describes authentication and authorization mechanisms. Multi-tenancy describes how multiple tenants share the repository.

Reliability properties describe data protection. Durability describes how data survives failures. Backup and recovery describes data protection mechanisms. Replication describes how data is copied for availability.

2.2 Data Accessor Components

Data accessor components are the clients of the repository, reading from and writing to the shared data store. They implement business logic, user interfaces, or integration functions that depend on persistent data.

2.2.1 Types of Data Accessor Components

Data accessor components vary in their relationship to data.

Data producers primarily write data to the repository. They may create new records, update existing data, or delete obsolete data. Examples include data entry applications, ETL processes, and event collectors.

Data consumers primarily read data from the repository. They query data for display, analysis, or processing. Examples include reporting tools, analytics applications, and data-driven user interfaces.

Data transformers both read and write, transforming data in place or producing derived data. Examples include data quality processes, aggregation jobs, and materialized view maintenance.

Interactive applications combine reading and writing in response to user actions. Most business applications fall into this category.

Background processes access data autonomously, performing maintenance, synchronization, or scheduled processing.

2.2.2 Essential Properties of Data Accessor Components

Accessor properties include access patterns describing whether the component primarily reads, writes, or both; query patterns describing the types of queries issued; data scope describing which portions of the data are accessed; concurrency describing how many instances may access simultaneously; transaction requirements specifying ACID requirements for data operations; and performance requirements specifying latency and throughput needs.

2.3 Data Reading and Writing Connectors

Connectors in the shared-data style enable communication between data accessors and repositories. They provide the mechanisms for reading, writing, and querying data.

2.3.1 Types of Connectors

Connectors vary in their capabilities and protocols.

Database connectors use database protocols and APIs. JDBC provides Java database connectivity. ODBC provides open database connectivity for various languages. Native drivers provide database-specific optimized access. ORM frameworks like Hibernate and Entity Framework provide object-relational mapping.

File connectors provide access to file-based storage. File system APIs provide direct file access. Network file protocols like NFS and SMB provide remote file access. Object storage APIs like S3 provide cloud object access.

Query connectors support specific query languages. SQL connectors support relational queries. GraphQL connectors support graph queries. Full-text search connectors support text search. Custom query languages may be specific to particular repositories.

Streaming connectors support continuous data flow. Change data capture streams database changes. Event streaming platforms like Kafka provide pub-sub over data. Real-time sync keeps multiple data stores synchronized.

2.3.2 Essential Properties of Connectors

An important property is whether the connector is transactional or not.

Transaction properties describe consistency guarantees. Transactional support indicates whether ACID transactions are supported. Isolation levels specify what isolation guarantees are provided. Distributed transactions indicate whether transactions can span multiple repositories.

Communication properties describe connection behavior. Synchronous vs. asynchronous indicates whether operations block or return immediately. Connection management describes pooling, timeouts, and retry behavior. Batching indicates whether multiple operations can be batched.

Performance properties characterize connector efficiency. Latency describes communication delay. Throughput describes maximum operation rate. Caching describes client-side caching capabilities.

Security properties address data protection. Encryption indicates whether connections are encrypted. Authentication describes how accessors prove identity. Authorization describes how permissions are enforced.

3 Relations

Relations in the shared-data style define how data accessors connect to repositories and how data flows through the system.

3.1 Attachment Relation

The *attachment* relation determines which data accessors are connected to which data repositories. This relation defines the data access topology of the system.

3.1.1 Properties of Attachment

Access mode specifies whether the attachment supports reading, writing, or both.

Connector type specifies which connector is used for the attachment.

Scope specifies which portions of the repository data are accessible through this attachment.

Permissions specify what operations are authorized through this attachment.

3.2 Reads-From Relation

The *reads-from* relation indicates that a data accessor reads data from a repository. This relation captures data consumption dependencies.

3.2.1 Properties of Reads-From

Query patterns describe the types of queries issued.

Read frequency describes how often reads occur.

Data scope describes which data is read.

Consistency requirements specify how current data must be.

3.3 Writes-To Relation

The *writes-to* relation indicates that a data accessor writes data to a repository. This relation captures data production dependencies.

3.3.1 Properties of Writes-To

Write patterns describe the types of writes (insert, update, delete).

Write frequency describes how often writes occur.

Data scope describes which data is written.

Transaction requirements specify consistency requirements for writes.

3.4 Data Flow Relation

The *data flow* relation describes how data moves through the system via the shared repository. Data flows from producers to the repository to consumers.

Understanding data flow is essential for analyzing system behavior, performance, and consistency.

3.5 Schema Dependency Relation

The *schema dependency* relation indicates that a data accessor depends on specific aspects of the repository schema. Schema changes may require accessor changes.

3.5.1 Properties of Schema Dependency

Dependency scope specifies which schema elements are depended upon.

Coupling strength indicates how tightly the accessor is coupled to schema details.

Version tolerance indicates whether the accessor can handle schema variations.

4 Computational Model

The computational model describes how shared-data systems execute.

4.1 Data-Mediated Communication

Communication between data accessors is mediated by a shared-data store. Rather than components sending messages directly to each other, they communicate indirectly through the data they share.

A producer component writes data to the repository. The repository persists the data. A consumer component reads the data from the repository. The producer and consumer need not be aware of each other.

This indirect communication provides temporal decoupling (producer and consumer need not be active simultaneously), identity decoupling (producer and consumer need not know each other's identity), and interface decoupling (producer and consumer share only the data model).

4.2 Control Models

Control may be initiated by data accessors or by the data store.

In accessor-initiated control (repository model), data accessors decide when to read and write. The repository is passive, responding to requests. Polling may be needed to detect data changes. This model is simpler but may miss timely updates.

In repository-initiated control (blackboard model), the data store notifies accessors of relevant changes. Accessors register interest in specific data. The repository triggers accessors when data changes. This model provides timely response but requires notification infrastructure.

Hybrid models combine both approaches. Accessors initiate most operations. Notifications alert accessors to important changes. This balances simplicity with responsiveness.

4.3 Data Persistence

Data is made persistent by the data store. Persistence ensures data survives process and system failures.

Persistence mechanisms vary by repository type. Write-ahead logging records changes before applying them. Checkpointing periodically saves consistent state. Replication maintains multiple copies. Journaling tracks changes for recovery.

Persistence guarantees range from best-effort to durable. Synchronous persistence confirms data is durable before returning. Asynchronous persistence returns before durability is confirmed. Configurable persistence lets applications choose based on requirements.

4.4 Concurrency and Consistency

Multiple accessors may read and write concurrently, requiring concurrency control.

Pessimistic concurrency uses locks to prevent conflicts. Readers may block writers. Writers may block readers and other writers. Deadlock detection and resolution may be needed.

Optimistic concurrency allows concurrent access and detects conflicts at commit. Version checking identifies conflicting updates. Conflicts require application-level resolution.

Multi-version concurrency control maintains multiple versions of data. Readers see consistent snapshots without blocking writers. Old versions are garbage collected.

Consistency models define what readers see relative to writes. Strong consistency ensures reads see the latest write. Eventual consistency allows temporary inconsistency with eventual convergence. Causal consistency preserves causal relationships between operations.

4.5 Transactions

Transactions group multiple operations into atomic units.

ACID transactions provide atomicity (all or nothing), consistency (valid state to valid state), isolation (concurrent transactions appear serial), and durability (committed changes persist).

Distributed transactions coordinate across multiple repositories. Two-phase commit ensures all-or-nothing across participants. Saga patterns decompose into local transactions with compensation.

Transaction isolation levels trade consistency for concurrency. Read uncommitted allows dirty reads. Read committed prevents dirty reads. Repeatable read prevents non-repeatable reads. Serializable provides full isolation.

5 Constraints

The shared-data style imposes constraints that define valid architectural configurations.

5.1 Access Constraints

Data accessors interact with the data store(s). All data access flows through the repository. Direct accessor-to-accessor data transfer bypasses the shared-data model.

5.2 Schema Constraints

Accessors must conform to repository schemas. Data written must match expected structure. Queries must reference valid schema elements. Schema changes may require accessor updates.

5.3 Consistency Constraints

The system must maintain data consistency. Integrity constraints enforce valid data states. Referential integrity maintains relationships. Business rules ensure semantic validity.

5.4 Concurrency Constraints

Concurrent access must be managed. Isolation requirements determine acceptable interference. Deadlock must be prevented or resolved. Resource exhaustion must be avoided.

5.5 Capacity Constraints

Repositories have finite capacity. Storage limits bound data volume. Connection limits bound concurrent accessors. Throughput limits bound operation rates.

5.6 Security Constraints

Data access must be controlled. Authentication verifies accessor identity. Authorization controls what data can be accessed. Audit logging tracks data access. Encryption protects sensitive data.

6 What the Style is For

The shared-data style supports several important architectural goals.

6.1 Multiple Component Access to Persistent Data

The primary purpose is allowing multiple components to access persistent data. This enables data sharing where multiple applications use the same data without duplication. It provides data integration by consolidating data from multiple sources. It ensures data consistency by maintaining single source of truth. It enables data longevity since data persists beyond individual component lifetimes.

6.2 Decoupling Data Producers from Consumers

The style provides enhanced modifiability by decoupling data producers from data consumers.

Producers can write data without knowing who will read it. Consumers can read data without knowing who produced it. New consumers can be added without changing producers. New producers can be added without changing consumers. The data model provides a stable interface between producers and consumers.

This decoupling enables independent development where producer and consumer teams work independently. It supports independent deployment since components can be deployed separately. It provides technology flexibility since producers and consumers can use different technologies. It enables temporal decoupling since producers and consumers need not be active simultaneously.

6.3 Enabling Data Analytics

Shared repositories enable analytical processing.

Data warehouses consolidate data for business intelligence. Data lakes store raw data for diverse analyses. Data marts provide focused analytical datasets. Reporting tools query shared analytical stores.

6.4 Supporting Integration

Shared data provides an integration mechanism.

Data-based integration connects systems through shared data. ETL processes move data between systems. Change data capture propagates changes. Master data management maintains shared reference data.

6.5 Enabling Collaboration

Shared data enables collaborative work.

Multiple users access shared documents or records. Changes by one user are visible to others. Conflict resolution handles concurrent changes. Audit trails track who changed what.

7 Notations

Shared-data architectures can be represented using various notations.

7.1 Data Flow Diagrams

Data flow diagrams show how data moves through the system. Data stores are shown as open-ended rectangles or parallel lines. Processes (accessors) are shown as circles or rounded rectangles. Data flows are shown as arrows between elements.

7.2 Entity-Relationship Diagrams

ER diagrams document repository data models. Entities represent data types. Relationships show connections between entities. Attributes detail entity properties. Cardinality specifies relationship multiplicities.

7.3 UML Component Diagrams

UML can represent shared-data architectures. Components represent repositories and accessors. Dependencies show access relationships. Interfaces represent data access contracts.

7.4 UML Deployment Diagrams

Deployment diagrams show physical data distribution. Nodes represent servers or storage systems. Artifacts represent deployed databases. Communication paths show data access connections.

7.5 Architecture Description Languages

Formal ADLs can specify shared-data architectures. Data stores are formally defined. Access relationships are explicit. Quality attributes can be specified.

7.6 Data Modeling Notations

Various notations document data structure. Chen notation provides classic ER modeling. Crow's foot notation shows cardinality graphically. UML class diagrams model data as classes. JSON Schema and XML Schema define document structures.

8 Quality Attributes

Shared-data decisions significantly affect system quality attributes.

8.1 Performance

Performance in shared-data systems depends on repository and access efficiency.

Read performance depends on indexing, caching, and query optimization. Proper indexes enable efficient data location. Query optimization plans efficient access paths. Caching reduces repeated access to storage.

Write performance depends on durability and consistency requirements. Synchronous writes ensure durability but add latency. Indexes must be updated on writes. Constraint checking adds overhead.

Concurrency affects performance as locks and isolation create overhead. Connection management affects throughput. Batching amortizes overhead across operations.

Performance tactics include query optimization and indexing, read replicas for read scaling, caching at multiple levels, connection pooling, and denormalization for read performance.

8.2 Scalability

Scalability addresses growing data and access demands.

Vertical scaling adds resources to a single repository. It is simpler to implement but has ultimate limits.

Horizontal scaling distributes data across multiple nodes. Sharding partitions data by key. Replication copies data for read scaling. Federation distributes by function.

Scalability considerations include data partitioning strategy, cross-partition queries, distributed transactions, and rebalancing as data grows.

8.3 Availability

Availability ensures data is accessible when needed.

Redundancy replicates data across nodes. Failover switches to backup when primary fails. Load balancing distributes access across replicas.

Availability tactics include replication for redundancy, automatic failover, health checking and monitoring, graceful degradation when partially available, and multi-region deployment.

The CAP theorem constrains distributed repositories. Consistency, availability, and partition tolerance cannot all be fully achieved. Systems must choose which to prioritize during partitions.

8.4 Security

Security protects data from unauthorized access and modification.

Authentication verifies accessor identity through credentials, tokens, or certificates.

Authorization controls data access through role-based access control, attribute-based access control, and row-level security.

Encryption protects data in transit through TLS and at rest through storage encryption.

Audit logging tracks data access for compliance and forensics.

Data masking protects sensitive data from unauthorized viewing.

8.5 Modifiability

The style supports modifiability through data-centric decoupling.

Schema evolution changes data structure over time. Backward-compatible changes add without removing. Migration handles incompatible changes. Versioning supports multiple schemas.

Accessor independence enables changing accessors without affecting others. New accessors can be added easily. Accessor technology can change independently.

Repository changes may affect all accessors. Schema changes may require accessor updates. Repository replacement is a major undertaking.

8.6 Reliability

Reliability ensures correct data storage and retrieval.

Data integrity enforces valid data through constraints, validation, and referential integrity.

Transaction reliability ensures atomic, consistent operations through ACID properties and recovery mechanisms.

Fault tolerance handles failures through redundancy, recovery, and error handling.

8.7 Testability

The style supports testability through data visibility.

Data verification checks stored data against expectations.

Test data management creates, manages, and cleans test data.

State inspection examines repository state during testing.

Data isolation separates test data from production.

9 Common Shared-Data Patterns

Several recurring patterns address common shared-data challenges.

9.1 Single Database Pattern

All accessors share a single database instance.

Benefits include simplicity with one database to manage, strong consistency with ACID transactions, and straightforward querying with joins across all data.

Limitations include scalability bounds at single-node limits, single point of failure, and schema coupling affecting all accessors.

This pattern is appropriate for smaller applications, when strong consistency is required, and when operational simplicity is valued.

9.2 Database-per-Service Pattern

Each service or bounded context has its own database.

Benefits include independent scaling of each database, technology flexibility per service, isolated failures, and team autonomy over their data.

Limitations include cross-service queries requiring coordination, distributed transactions being complex, and data consistency requiring explicit design.

This pattern is common in microservices architectures where services own their data.

9.3 Shared Database Pattern

Multiple services share a single database with separate schemas or access.

Benefits include simpler querying across service data, familiar ACID transactions, and lower operational overhead than multiple databases.

Limitations include schema coupling across services, potential performance interference, and coordination needed for schema changes.

This pattern provides a middle ground between full sharing and full separation.

9.4 CQRS Pattern

Command Query Responsibility Segregation separates read and write models.

Commands write to a write-optimized store. Queries read from read-optimized stores. Synchronization keeps read stores updated.

Benefits include independent optimization of reads and writes, read scaling through replicas, and query model tailored to query needs.

Limitations include complexity of maintaining multiple models, eventual consistency between models, and more infrastructure to manage.

9.5 Event Sourcing with Shared Log

State changes are stored as events in a shared log.

Events are appended to an immutable log. Current state is derived by replaying events. Multiple consumers read from the shared log.

Benefits include complete audit trail, ability to replay history, multiple consumers with different projections, and temporal queries.

Limitations include storage growth over time, replay time for reconstruction, and eventual consistency for derived state.

9.6 Data Lake Pattern

Raw data is stored in a central lake for diverse analytical access.

Data is ingested in native formats. Schema-on-read interprets data at query time. Multiple tools access the lake for different analyses.

Benefits include flexibility for diverse data types, raw data preservation, and support for varied analytical tools.

Limitations include potential for data swamp without governance, schema-on-read complexity, and query performance challenges.

9.7 Cache-Aside Pattern

Applications manage a cache alongside the primary repository.

Applications check cache before querying the repository. Cache misses load from the repository and populate the cache. Cache invalidation removes stale entries.

Benefits include improved read performance, reduced repository load, and application control over caching.

Limitations include cache invalidation complexity, potential staleness, and additional infrastructure.

9.8 Materialized View Pattern

Pre-computed query results are stored for efficient access.

Views are computed from base data. Views are refreshed on a schedule or trigger. Queries access views instead of computing from base data.

Benefits include query performance for complex computations, reduced load on base data, and consistent view of computed results.

Limitations include refresh latency causing staleness, storage overhead, and refresh computation cost.

9.9 Data Mesh Pattern

Data ownership is distributed to domain teams.

Each domain owns and serves its data as products. Federated governance provides interoperability. Self-serve infrastructure enables domain teams.

Benefits include domain expertise applied to data, scalable organization, and reduced central bottlenecks.

Limitations include coordination overhead, infrastructure complexity, and governance challenges.

10 Implementation Considerations

Implementing shared-data systems involves several practical considerations.

10.1 Data Modeling

Data models significantly affect system behavior.

Normalization reduces redundancy and update anomalies but may require joins for queries. Denormalization improves read performance but increases redundancy and update complexity.

Schema design should balance current and future needs. Overly specific schemas are hard to evolve. Overly generic schemas are hard to query efficiently.

Data types should match domain semantics. Proper types enable validation and optimization. Type mismatches cause bugs and performance issues.

10.2 Index Design

Indexes critically affect query performance.

Primary indexes enable efficient key-based access. Secondary indexes support queries on non-key columns. Composite indexes support multi-column queries. Full-text indexes enable text search. Spatial indexes support geographic queries.

Index trade-offs balance read and write performance. More indexes improve read performance. More indexes slow write performance. Index selection requires understanding query patterns.

10.3 Partitioning Strategies

Partitioning distributes data across multiple stores.

Horizontal partitioning (sharding) distributes rows across partitions. Range partitioning groups data by value ranges. Hash partitioning distributes by hash of partition key. Directory partitioning uses a lookup for partition assignment.

Vertical partitioning distributes columns across partitions. Frequently accessed columns are grouped. Large columns may be stored separately.

Partitioning considerations include choosing a partition key that distributes evenly, handling cross-partition queries, and rebalancing as data grows.

10.4 Replication Strategies

Replication copies data for availability and read scaling.

Synchronous replication confirms writes on all replicas. It provides strong consistency but adds latency.

Asynchronous replication confirms writes before replica update. It provides lower latency but allows stale reads.

Replication topologies include primary-replica, multi-primary, and peer-to-peer arrangements.

Conflict resolution handles concurrent updates to replicas through last-write-wins, vector clocks, or custom resolution.

10.5 Connection Management

Connections between accessors and repositories require management.

Connection pooling reuses connections across requests, reducing connection establishment overhead.

Connection limits bound concurrent connections to prevent resource exhaustion.

Timeout configuration balances responsiveness against premature failures.

Retry policies handle transient failures without overwhelming the repository.

10.6 Data Migration

Data must be migrated for schema changes and repository changes.

Schema migrations update database structure through versioned migration scripts. Backward-compatible migrations allow incremental deployment. Breaking migrations require coordinated deployment.

Data migrations move data between repositories. ETL processes extract, transform, and load. Change data capture enables incremental migration. Dual-write patterns enable gradual cutover.

11 Examples

Concrete examples illustrate shared-data concepts.

11.1 Enterprise Application

A traditional enterprise application uses shared-data architecture.

A relational database stores business data including customers, orders, and products.

Multiple applications access the database. Web applications provide user interfaces. Batch jobs process overnight tasks. Reporting tools generate business reports. Integration processes exchange data with partners.

Benefits include single source of truth, transactional consistency, and familiar SQL access.

Challenges include database becoming a bottleneck, tight coupling through shared schema, and all changes requiring coordination.

11.2 Data Warehouse

A data warehouse consolidates data for analytics.

ETL processes extract from operational systems, transform for analytical use, and load into the warehouse.

Multiple tools access the warehouse. BI tools generate reports and dashboards. Analysts query with SQL. Data scientists extract data for models.

Dimensional modeling organizes data for analytical queries. Fact tables contain measures. Dimension tables contain descriptive attributes.

Benefits include consolidated analytical view, optimized for analytical queries, and historical data preservation.

11.3 Content Management System

A CMS uses shared-data for content.

A content repository stores articles, media, and metadata.

Content producers create and edit content through authoring tools.

Content consumers access content through websites, APIs, and mobile apps.

Workflow systems manage content lifecycle from draft through review to publication.

Benefits include content reuse across channels, collaborative authoring, and version control.

11.4 Distributed Cache

A distributed cache improves application performance.

A caching layer sits between applications and primary storage.

Applications check cache before querying the database. Cache hits return immediately. Cache misses query the database and populate the cache.

Multiple application instances share the cache, providing consistent cached data.

Benefits include reduced database load, improved response time, and shared cache efficiency.

Technologies include Redis, Memcached, and Hazelcast.

11.5 Data Lake Analytics

A data lake supports diverse analytics.

Data is ingested from multiple sources in native formats, including structured database exports, semi-structured logs and events, and unstructured documents and media.

Multiple analytical tools access the lake. SQL engines query structured data. Spark processes large-scale analytics. ML platforms train models. BI tools visualize results.

Data governance maintains data quality and security through catalogs documenting available data, access controls protecting sensitive data, and lineage tracking data origins.

Benefits include flexible schema, diverse analytical support, and raw data preservation.

11.6 Microservices with Shared Database

A microservices system shares a database with careful design.

Services share a database but own distinct schemas or tables.

Clear ownership assigns each table to one service. Other services access through the owning service's API or through shared read-only views.

Benefits include simpler querying than separate databases, familiar transactional behavior, and lower operational overhead.

Challenges include maintaining ownership discipline and coordinating schema changes.

12 Best Practices

Experience suggests several best practices for shared-data systems.

12.1 Design Data Models Carefully

Data models are hard to change and affect all accessors.

Invest time in understanding domain data. Normalize appropriately for the use case. Plan for evolution through extensible designs. Document models thoroughly.

12.2 Establish Clear Data Ownership

Ownership clarity prevents conflicts and enables maintenance.

Assign each data entity to an owning team or service. The owner controls schema evolution. Other accessors coordinate with owners. Ownership should align with business responsibility.

12.3 Manage Schema Evolution

Schemas will change; plan for it.

Use versioned migration scripts. Prefer backward-compatible changes. Coordinate breaking changes across accessors. Test migrations thoroughly.

12.4 Implement Appropriate Consistency

Choose consistency levels based on requirements.

Strong consistency where correctness requires it. Eventual consistency where performance requires it. Document consistency assumptions and implications.

12.5 Secure Data Appropriately

Data security requires deliberate design.

Implement authentication for all access. Authorize based on least privilege. Encrypt sensitive data. Audit data access.

12.6 Monitor Data Systems

Visibility enables effective operation.

Monitor query performance. Track storage growth. Alert on errors and anomalies. Measure against SLAs.

12.7 Plan for Scale

Growth is inevitable; plan for it.

Choose scalable repository technologies. Design for horizontal scaling. Monitor capacity and plan upgrades.

12.8 Test Data Operations

Data operations require thorough testing.

Test with realistic data volumes. Test concurrent access patterns. Test failure and recovery scenarios. Test migration procedures.

13 Common Challenges

Shared-data systems present several common challenges.

13.1 Schema Coupling

Shared schemas couple all accessors.

Schema changes may break multiple accessors. Evolution requires coordination across teams. Technical debt accumulates in schemas.

Strategies include clear ownership, interface abstraction through views and services, backward-compatible evolution, and schema versioning.

13.2 Performance Bottlenecks

Shared repositories can become bottlenecks.

All accessors compete for repository resources. Hot spots concentrate load. Growth outpaces capacity.

Strategies include caching to reduce load, read replicas for read scaling, partitioning to distribute load, query optimization, and capacity planning.

13.3 Consistency vs. Availability Trade-offs

The CAP theorem forces trade-offs.

Strong consistency may sacrifice availability. High availability may allow inconsistency. Network partitions force choices.

Strategies include choosing appropriate consistency per operation, designing for partition handling, and using patterns like saga for distributed consistency.

13.4 Data Quality

Shared data quality affects all consumers.

Garbage in, garbage out affects downstream processes. Multiple producers may introduce inconsistency. Quality issues are expensive to fix after the fact.

Strategies include validation at ingestion, data quality monitoring, clear data ownership, and master data management.

13.5 Security and Compliance

Shared data creates security challenges.

Broad access increases exposure risk. Compliance requirements constrain data handling. Audit requirements track access.

Strategies include least-privilege access, data classification, encryption, audit logging, and compliance automation.

13.6 Operational Complexity

Shared data systems require careful operation.

Backup and recovery protect against data loss. Monitoring detects issues early. Capacity management prevents exhaustion. Incident response addresses failures.

Strategies include automation where possible, runbooks for common tasks, regular testing of procedures, and investment in observability.

13.7 Legacy Data Systems

Legacy systems create challenges.

Outdated technology limits capabilities. Accumulated technical debt slows progress. Institutional knowledge may be lost. Migration is risky and expensive.

Strategies include incremental modernization, abstraction layers, documentation investment, and careful migration planning.

14 Conclusion

The shared-data style provides a powerful pattern for building systems around persistent data repositories. By mediating component communication through shared data stores, the style enables data sharing, decoupling, and persistence that support many types of applications.

Effective shared-data architecture requires attention to data modeling, consistency requirements, performance optimization, and operational concerns. The patterns and practices described in this document provide guidance for building robust, scalable, and maintainable data-centric systems.

The style remains fundamental to software architecture, from traditional database applications to modern data platforms. Understanding shared-data architecture equips architects to design effective systems where persistent data is central to system function and value.

As data volumes grow and analytical demands increase, the shared-data style continues to evolve through distributed databases, data lakes, and data mesh patterns. The fundamental insight—that shared persistent data can mediate component interaction—remains relevant across this evolution.

References

- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley Professional.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
- Date, C. J. (2019). *Database Design and Relational Theory* (2nd ed.). Apress.
- Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit* (3rd ed.). Wiley.
- Dehghani, Z. (2022). *Data Mesh: Delivering Data-Driven Value at Scale*. O'Reilly Media.
- Sadalage, P. J., & Fowler, M. (2012). *NoSQL Distilled*. Addison-Wesley Professional.