Prev years Qs on papers

D1 [5].[Kafka] What kind of ordering guarantees are provided by Kafka? What kind of ordering cannot be guaranteed and why? D2 [5].

D1: Kafka's Ordering Guarantees

Kafka provides specific ordering guarantees for messages within its system:

- **Ordering within a Partition**: Kafka guarantees message ordering within a single partition. Messages sent by a producer to a particular topic partition will be appended in the order they are sent. This means if a record M1 is sent before a record M2 by the same producer, M1 will have a lower offset than M2 and will appear earlier in the log[3][4][5].

- **Partition Key**: Ordering is preserved for messages with the same key because these messages are routed to the same partition, and consumers read messages in order from each partition[4].

Ordering Limitations:

- **Cross-Partition Ordering**: Kafka does not guarantee ordering across different partitions within the same topic. Each partition operates independently, so messages from different partitions can be processed out of order relative to each other[4][5].

- **Producer Retries**: If a producer fails to receive an acknowledgment for a message and retries, it can lead to an out-of-order scenario if the original attempt was successful but the acknowledgment was lost. However, Kafka provides an idempotent option to prevent duplicate entries in the log, ensuring log order is maintained, but this does not cover cases where the producer sends messages in a different order upon retry[6].

- **Partition Count Changes**: Changing the number of partitions for a topic after it has been in production can result in a loss of ordering, as keys might be assigned to new partitions, disrupting the order of messages with the same key[4].

[MapReduce] What are a) the actions taken by the master in response to a worker failure b) the implications for both map and reduce tasks in the event of a worker failure

D2: MapReduce and Worker Failure

a) Actions Taken by the Master in Response to Worker Failure:

- **Map Task Failure**: If a worker fails while executing map tasks, the master:
  - Reassigns all uncompleted map tasks to other idle workers.
  - Re-executes any completed map tasks that were not successfully committed (e.g., if the worker failed before it could communicate its completion).

- **Reduce Task Failure**: If a worker fails during reduce tasks:
  - The master reassigns the reduce task to another available worker.

- **General Actions**: The master periodically checks the health of workers by sending "ping" messages. If a worker fails to respond to several pings, it is considered dead, and its tasks are reassigned[5].

b) Implications for Map and Reduce Tasks in the Event of a Worker Failure:

- **Map Tasks**:

  - If a worker fails, any map tasks assigned to it must be re-executed. This can cause:

    - **Rerun of Map Tasks**: All map tasks not completed or not successfully committed by the failed worker must be rerun, potentially increasing the overall job execution time.

    - **Data Loss**: If the failed worker had completed map tasks but not committed them, those outputs are lost, leading to recomputation.

- **Reduce Tasks**:

  - **Data Integrity**: Since reduce tasks depend on the outputs of map tasks, any data produced by a failed worker's map tasks must be recomputed or fetched from other workers' completed map outputs.

  - **Delay**: The reassignment of reduce tasks can delay the overall job completion as the new worker needs to fetch the necessary data from other workers or recompute, if necessary.

- **Overall System**:

  - **Fault Tolerance**: The system ensures fault tolerance by reassigning tasks, but this comes at the cost of increased resource utilization and potential delays.

  - **Scalability**: The ability to reassign tasks across workers allows the system to scale by adding more workers to handle the load, enhancing the overall robustness and efficiency of the MapReduce job execution process.

GFS: Using Fig 1, describe how a chuck server failure is handled in GFS

GFS: Chunk Server Failure Handling

GFS (Google File System) is designed to handle chunk server failures gracefully. While the image mentioned is not available in the search results, I can explain the general process:

1. **Replication**: GFS maintains multiple replicas (typically 3) of each chunk across different chunk servers.
2. **Heartbeat Mechanism**: The master periodically communicates with chunk servers through heartbeat messages.
3. **Failure Detection**: If a chunk server fails to respond to heartbeat messages for a certain period, the master marks it as down.
4. **Re-replication**: The master initiates the re-replication process for chunks that were stored on the failed server:
    - It identifies under-replicated chunks
    - Prioritizes re-replication based on factors like the number of available replicas
    - Instructs other chunk servers to create new copies of the affected chunks
5. **Metadata Update**: The master updates its metadata to reflect the new chunk locations.
6. **Client Redirection**: Clients requesting data from the failed chunk server are redirected to other servers holding replicas of the requested chunks.

This approach ensures data availability and system reliability even in the face of chunk server failures.

GFS: Why single GFS master is used? How does GFS recover from GFS master failure

GFS: Single Master and Master Failure Recovery

GFS uses a single master design for several reasons:

1. **Simplicity**: A single master simplifies the system architecture and coordination.

2. **Consistency**: It's easier to maintain a consistent view of the file system state.
3. **Efficient Metadata Management**: Centralized metadata management is more efficient for a distributed file system.

However, this design raises concerns about the master being a single point of failure. GFS addresses master failure recovery as follows:

1. **Shadow Masters**: GFS maintains shadow masters that replicate the master's state.
2. **Operation Logging**: All metadata-changing operations are logged to multiple remote machines.
3. **Checkpointing**: The master's state is periodically checkpointed to facilitate faster recovery.
4. **Fast Recovery**: In case of master failure, a new master can be started quickly using the latest checkpoint and operation logs.
5. **Read-only Mode**: During master recovery, the system can still serve read requests using shadow masters.

This approach balances the benefits of a single master with robust failure recovery mechanisms.

BigQuery: What are key differences between BigQuery and MapReduce? Give examples of use cases where BigQuery is preferred over MapReduce and vice versa

BigQuery vs MapReduce: Key Differences and Use Cases

BigQuery and MapReduce are both powerful tools for processing large datasets, but they have significant differences124:

Key Differences:

Processing Model:

BigQuery: Uses a columnar storage system and executes SQL-like queries.

MapReduce: Uses a batch processing model with custom code.

Ease of Use:

BigQuery: Fully managed service, easier to set up and use.

MapReduce: Requires cluster setup and management, more complex.

Query Speed:

BigQuery: Designed for interactive, near real-time queries.

MapReduce: Better suited for long-running batch jobs.

Scalability:

BigQuery: Automatically scales to handle large datasets.

MapReduce: Requires manual scaling of the Hadoop cluster.

Pricing Model:

BigQuery: On-demand pricing based on data processed.

MapReduce: Typically involves fixed costs for maintaining a cluster.

Use Cases where BigQuery is Preferred:

Ad-hoc Analysis: BigQuery excels at quick, interactive queries on large datasets1.

Business Intelligence: Ideal for generating reports and dashboards with near real-time data1.

Data Exploration: Allows data scientists to quickly explore and analyze large datasets2.

Streaming Analytics: Can handle real-time data ingestion and analysis4.

Use Cases where MapReduce is Preferred:

Complex Data Transformations: Better suited for intricate data processing that requires custom code1.

Large-scale ETL: Efficient for extract, transform, load operations on massive datasets2.

Machine Learning: When combined with tools like Mahout, it's powerful for distributed machine learning tasks.

Graph Processing: More suitable for complex graph algorithms and network analysis.

In practice, many organizations use both technologies complementarily. For example, they might use MapReduce for initial data processing and transformations, then load the results into BigQuery for fast, interactive analysis

[GFS] Specify three most important considerations/rationale behind Google to design GFS. Also specify what design choices were made to address these in GFS

GFS: Key Considerations and Design Choices

The three most important considerations behind Google's design of GFS were:

Component Failures as Norm: Google recognized that in a large-scale distributed system, component failures are frequent and expected.

Large File Sizes: GFS was designed to handle files that are much larger than traditional file systems, often multi-gigabyte in size.

Append-heavy Workload: The system was optimized for workloads that primarily append data to existing files rather than overwriting existing data.

Design choices made to address these considerations:

For Component Failures:

Chunk Replication: GFS stores multiple replicas (typically 3) of each chunk across different chunk servers13.

Automatic Re-replication: The system automatically re-replicates data from failed servers to maintain the desired replication level1.

Fault Detection: The master uses heartbeat messages to detect chunk server failures quickly1.

For Large File Sizes:

Large Chunk Size: GFS uses a chunk size of 64 MB, much larger than traditional file systems12.

Efficient Metadata Management: The large chunk size reduces the amount of metadata the master needs to manage1.

For Append-heavy Workload:

Record Append Operation: GFS introduced a custom "record append" operation that allows concurrent appends to a file1.

Relaxed Consistency Model: GFS implements a consistency model that supports concurrent appends while maintaining data integrity

C.3 [Borg] Explain the key components in Borg based on Fig.1 in the Borg paper.

Borg: Key Components

While the specific Figure 1 from the Borg paper is not available in the search results, I can describe the key components typically found in Borg's architecture:

Borgmaster: The central controller that manages the entire cluster. It handles client requests, schedules tasks, and monitors the health of machines and tasks.

Borglet: An agent that runs on each machine in the cluster. It starts and stops tasks, restarts them if they fail, and reports the machine's state to the Borgmaster.

Scheduler: Part of the Borgmaster that assigns tasks to machines based on resource requirements and availability.

Paxos Store: A distributed storage system used to store and replicate the cluster's state, ensuring high availability of the Borgmaster.

Borgcfg: A command-line tool that allows users to submit jobs and interact with the Borg system.

Sigma: A web-based user interface for interacting with Borg, visualizing cluster state, and managing jobs.