



**Kampus
Merdeka**
INDONESIA JAYA

BIG DATA (TFC303)

Pertemuan 3 – Pola Big Data

ALIFIA REVAN PRANANDA

Department of Information Technology
Faculty of Engineering
Universitas Tidar

TODAY'S MATERIALS

- ✓ Design Pattern of Big Data
- ✓ Pola MapReduce



DESIGN PATTERN OF BIG DATA

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)

These design patterns are useful for building reliable, scalable, secure applications in the cloud. Each pattern describes the problem that the pattern addresses, considerations for applying the pattern, and an example based on Microsoft Azure.

There are three component needed in the big data development :

- **Data management**

Data management is the key element of cloud applications, and it influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for performance, scalability or availability.

- **Design and implementation**

Good design encompasses consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and scenarios.

- **Messaging**

The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally loosely coupled to maximize scalability.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)

Catalog of patterns:

1. Ambassador
2. Anti-corruption Layer
3. Asynchronous Request-Reply
4. Backends for Frontends
5. Bulkhead
6. Cache-Aside
7. Choreography
8. Circuit Breaker
9. Claim Check
10. Compensating Transaction
11. Competing Consumers
12. Compute Resource Consolidation
13. CQRS
14. Deployment Stamps
15. Edge Workload Configuration
16. Event Sourcing
17. External Configuration Store
18. Federated Identity
19. Gatekeeper
20. Gateway Aggregation
21. Gateway Offloading
22. Gateway Routing
23. Geode
24. Health Endpoint Monitoring
25. Index Table
26. Leader Election
27. Materialized View
28. Pipes and Filters
29. Priority Queue
30. Publisher/Subscriber
31. Queue-Based Load Leveling
32. Rate Limiting
33. Retry
34. Saga
35. Scheduler Agent Supervisor
36. Sequential Convoy
37. Sharding
38. Sidecar
39. Static Content Hosting
40. Strangler Fig
41. Throttling
42. Valet Key

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Queue-Based Load Leveling Pattern

In the big data development, we often face to the following problems:

- Many solutions in the cloud **involve running tasks that invoke services**. In this environment, if a service is subjected to intermittent heavy loads, it **can cause performance or reliability issues**.
- A service could be part of the same solution as the tasks that use it, or it could be a third-party service providing access to frequently used resources such as a cache or a storage service. If the **same service is used** by a number of tasks running concurrently, it can be **difficult to predict** the volume of requests to the service at any time.
- A service might experience peaks in demand that cause it to overload and be unable to respond to requests in a timely manner. **Flooding a service** with a large number of concurrent requests can also **result in the service failing** if it's unable to handle the contention these requests cause.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Queue-Based Load Leveling Pattern

Queue is significant for solving load management. The queue acts as a buffer, storing the message until it's retrieved by the service. The service retrieves the messages from the queue and processes them. Requests from a number of tasks, which can be generated at a highly variable rate, can be passed to the service through the same message queue. This figure shows using a queue to level the load on a service.



DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Queue-Based Load Leveling Pattern

This pattern provides the following benefits:

1. It can help to maximize availability because delays arising in services won't have an immediate and direct impact on the application, which can continue to post messages to the queue even when the service isn't available or isn't currently processing messages.
2. It can help to maximize scalability because both the number of queues and the number of services can be varied to meet demand.
3. It can help to control costs because the number of service instances deployed only have to be adequate to meet average load rather than the peak load.

IMPORTANT NOTES:

- ❖ This pattern **is useful** to any application that uses services that are subject to overloading.
- ❖ This pattern **isn't useful** if the application expects a response from the service with minimal latency.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Competing Consumers Pattern

The number of requests in big data management can vary significantly **over time for many reasons** for example:

- A **sudden increase** in user activity or aggregated requests coming from multiple tenants can cause an unpredictable workload.
- At peak hours, a system might need to process many hundreds of requests per second, while at other times the number could be very small → **fluctuated process**.
- Additionally, the nature of the work performed to handle these requests might be **highly variable**. By using a single instance of the consumer service, you can cause that instance to become flooded with requests. Or, the messaging system might be overloaded by an influx of messages that come from the application.

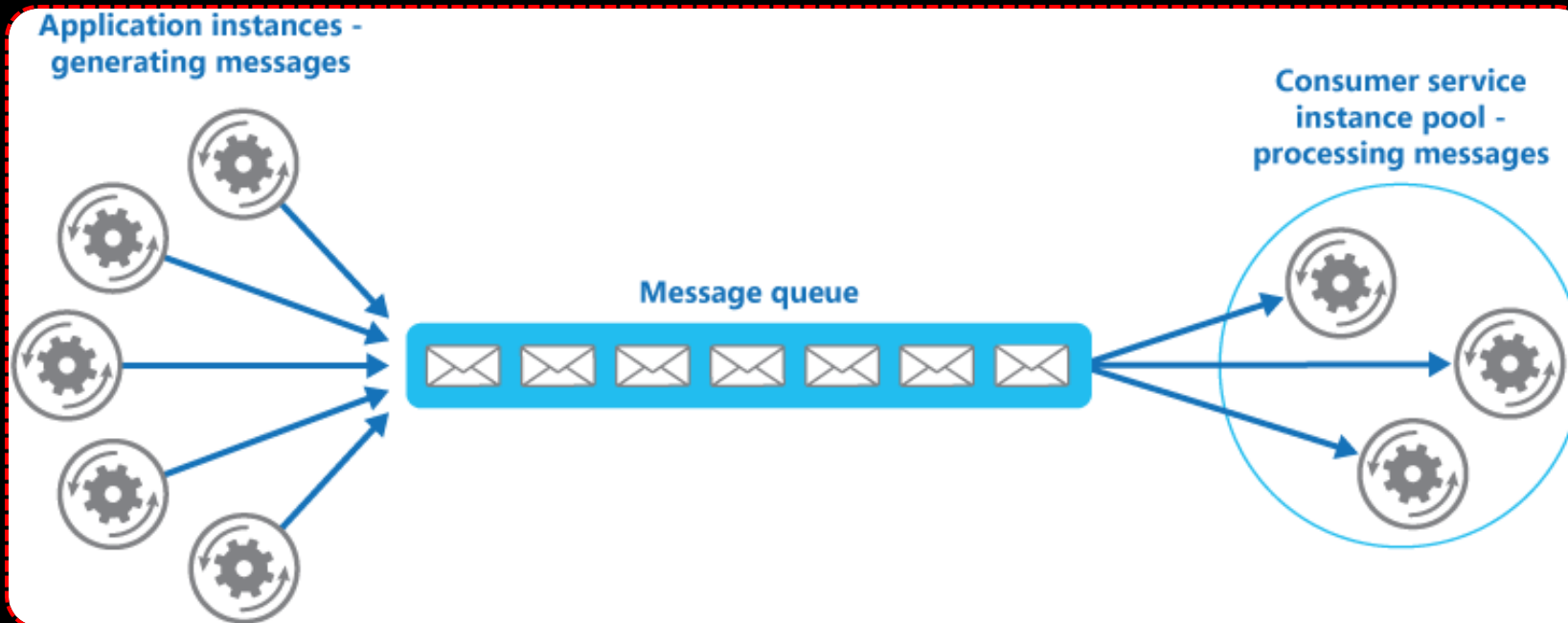
To handle this fluctuating workload, the system can **run multiple instances** of the consumer service. However, these consumers **must be coordinated** to ensure that each message is only delivered to a single consumer. The workload also needs to be load balanced across consumers to prevent an instance from becoming a bottleneck.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Competing Consumers Pattern

Use a message queue to implement the communication channel between the application and the instances of the consumer service. The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them. This approach enables the same pool of consumer service instances to handle messages from any instance of the application. The figure illustrates using a message queue to distribute work to instances of a service.



DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Competing Consumers Pattern

This pattern has the following benefits:

- It provides a load-leveled system that can handle wide variations in the volume of requests sent by application instances.
- It improves reliability.
- It doesn't require complex coordination between the consumers, or between the producer and the consumer instances. The message queue ensures that each message is delivered at least once.
- It's scalable. When you apply auto-scaling, the system can dynamically increase or decrease the number of instances of the consumer service as the volume of messages fluctuates.
- It can improve resiliency if the message queue provides transactional read operations. If a consumer service instance reads and processes the message as part of a transactional operation, and the consumer service instance fails, this pattern can ensure that the message will be returned to the queue to be picked up and handled by another instance of the consumer service. In order to mitigate the risk of a message continuously failing, we recommend you make use of dead-letter queues.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Competing Consumers Pattern

Use this pattern when:

- The workload for an application is divided into tasks that can run asynchronously.
- Tasks are independent and can run in parallel.
- The volume of work is highly variable, requiring a scalable solution.
- The solution must provide high availability, and must be resilient if the processing for a task fails.

This pattern might not be useful when:

- It's not easy to separate the application workload into discrete tasks, or there's a high degree of dependence between tasks.
- Tasks must be performed synchronously, and the application logic must wait for a task to complete before continuing.
- Tasks must be performed in a specific sequence.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Ambassador Pattern

Resilient cloud-based applications **require** features such as circuit breaking, routing, metering and monitoring, and the ability to make network-related configuration updates. It **may be difficult or impossible** to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.

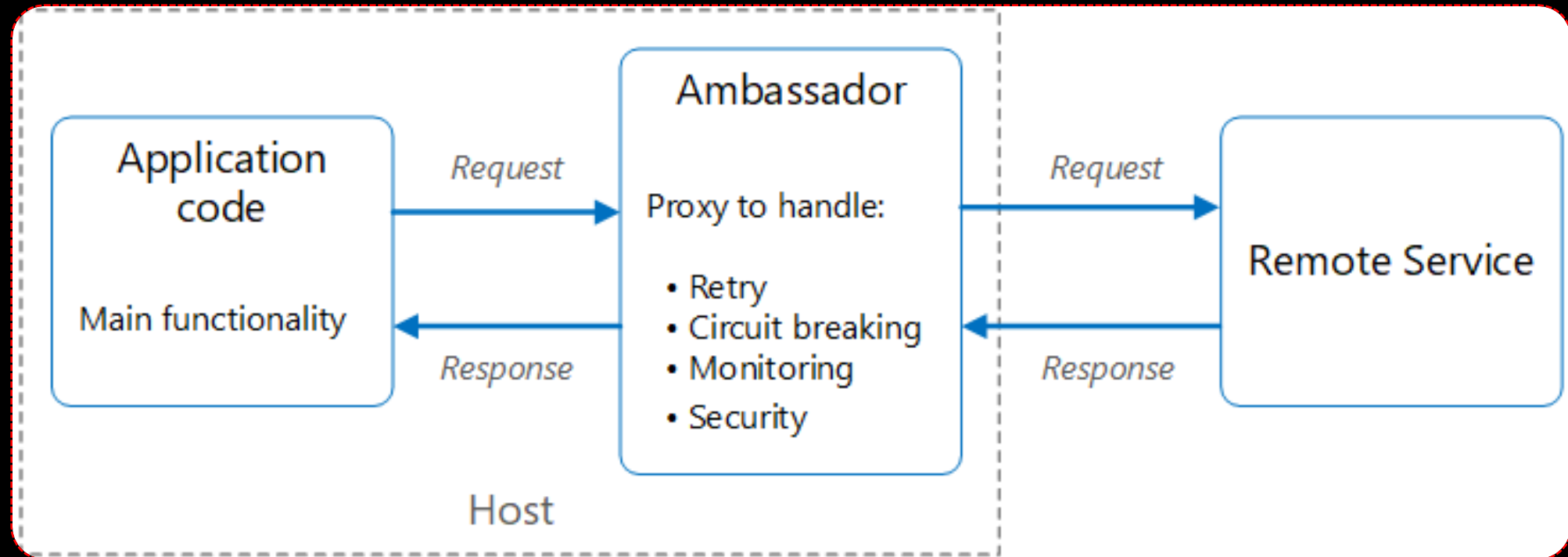
Network calls may **also require** substantial configuration for connection, authentication, and authorization. If these calls are used across multiple applications, built using multiple languages and frameworks, the calls must be configured for each of these instances. In addition, network and security functionality may need to be managed by a central team within your organization. With a large code base, it can be risky for that team to update application code they aren't familiar with.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Ambassador Pattern

You can use the ambassador pattern to standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.



DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Ambassador Pattern

Use this pattern when you:

- Need to build a common set of client connectivity features for multiple languages or frameworks.
- Need to offload cross-cutting client connectivity concerns to infrastructure developers or other more specialized teams.
- Need to support cloud or cluster connectivity requirements in a legacy application or an application that is difficult to modify.

This pattern may not be suitable:

- When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases this may affect the application.
- When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
- When connectivity features cannot be generalized and require deeper integration with the client application.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Sharding Pattern

A data store hosted by a single server might be subject to the following limitations:

- **Storage space** : A data store for a large-scale cloud application is expected to contain a huge volume of data that could increase significantly over time.
- **Computing resources** : A single server hosting the data store might not be able to provide the necessary computing power to support this load, resulting in extended response times for users and frequent failures as applications attempting to store and retrieve data time out.
- **Network bandwidth** : Ultimately, the performance of a data store running on a single server is governed by the rate the server can receive requests and send replies.
- **Geography** : It might be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access.

Important Notes:

Use this pattern when a data store is likely to need to scale beyond the resources available to a single storage node, or to improve performance by reducing contention in a data store.

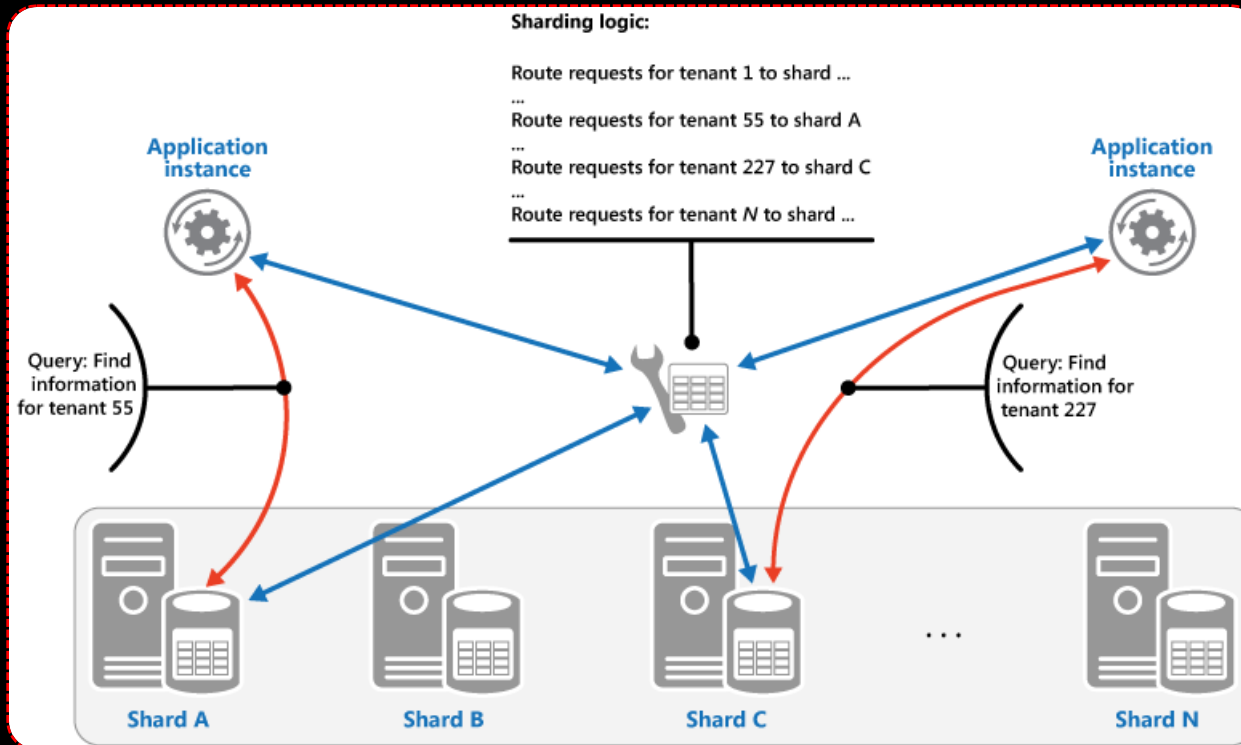
DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Sharding Pattern

Three strategies of sharding pattern are commonly used when selecting the shard key and deciding how to distribute data across shards. The strategies are:

1) The Lookup Strategy



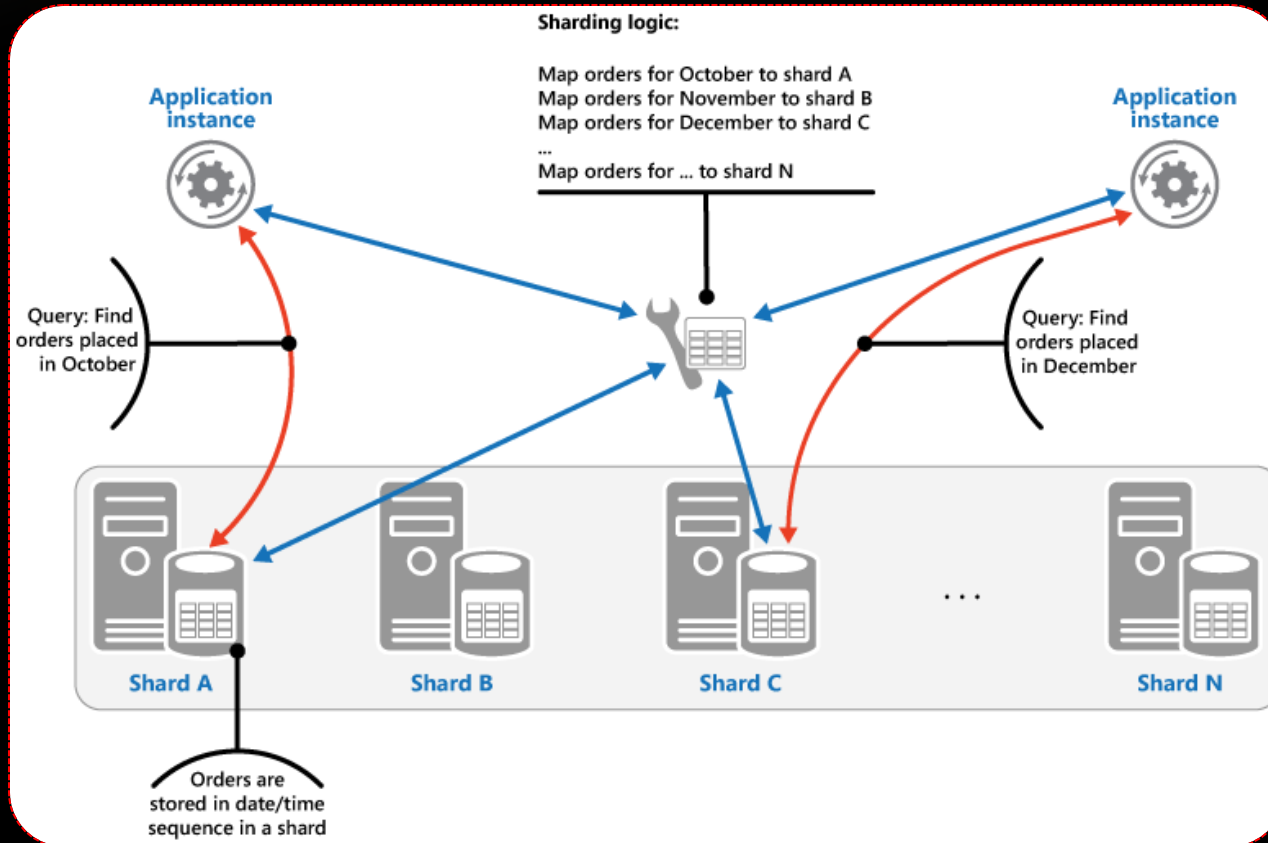
- In this strategy the sharding logic implements a map that routes a request for data to the shard that contains that data using the shard key.
- In a multi-tenant application all the data for a tenant might be stored together in a shard using the tenant ID as the shard key.
- Multiple tenants might share the same shard, but the data for a single tenant won't be spread across multiple shards.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Sharding Pattern

2) The Range Strategy



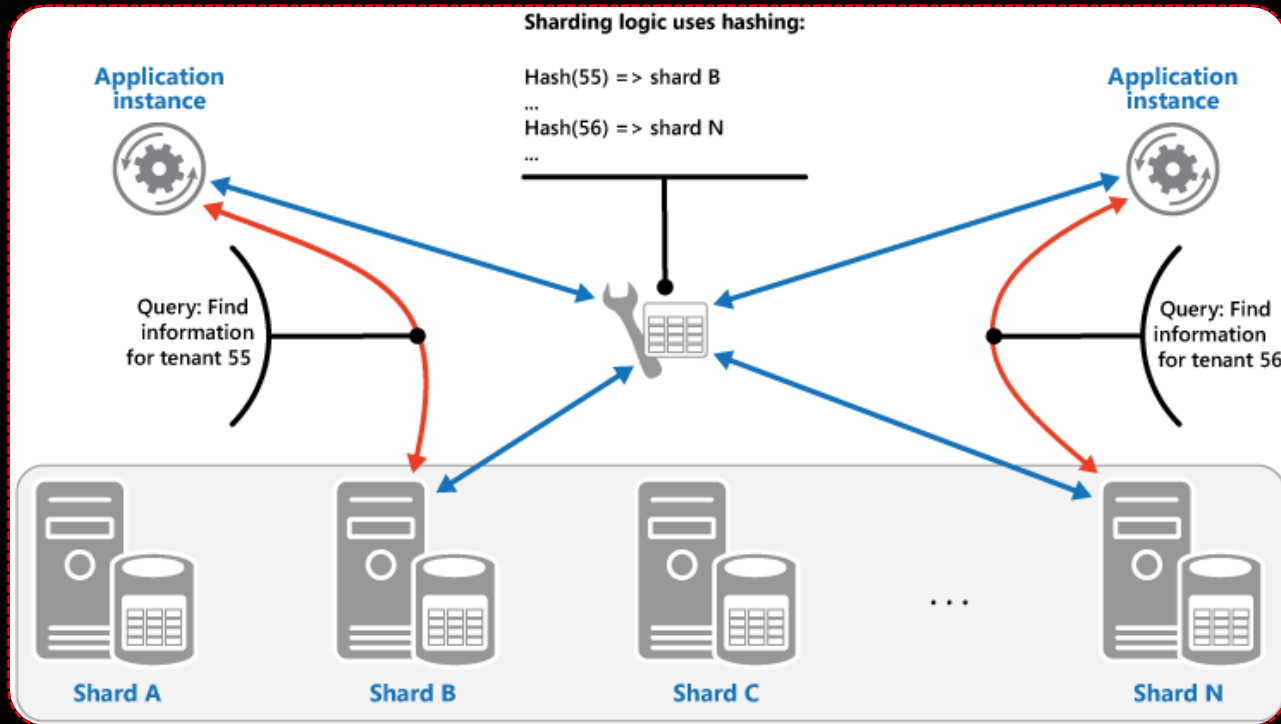
- This strategy groups related items together in the same shard, and orders them by shard key—the shard keys are sequential.
- It's useful for applications that frequently retrieve sets of items using range queries (queries that return a set of data items for a shard key that falls within a given range).

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Sharding Pattern

3) The Hash Strategy



- The purpose of this strategy is to reduce the chance of hotspots (shards that receive a disproportionate amount of load).
- It distributes the data across the shards in a way that achieves a balance between the size of each shard and the average load that each shard will encounter.
- The sharding logic computes the shard to store an item in based on a hash of one or more attributes of the data.
- The chosen hashing function should distribute data evenly across the shards, possibly by introducing some random element into the computation.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Sharding Pattern

The three sharding strategies have the following advantages and considerations:

- **Lookup.** This offers more control over the way that shards are configured and used. Using virtual shards reduces the impact when rebalancing data because new physical partitions can be added to even out the workload. The mapping between a virtual shard and the physical partitions that implement the shard can be modified without affecting application code that uses a shard key to store and retrieve data. Looking up shard locations can impose an additional overhead.
- **Range.** This is easy to implement and works well with range queries because they can often fetch multiple data items from a single shard in a single operation. This strategy offers easier data management. For example, if users in the same region are in the same shard, updates can be scheduled in each time zone based on the local load and demand pattern. However, this strategy doesn't provide optimal balancing between shards. Rebalancing shards is difficult and might not resolve the problem of uneven load if the majority of activity is for adjacent shard keys.
- **Hash.** This strategy offers a better chance of more even data and load distribution. Request routing can be accomplished directly by using the hash function. There's no need to maintain a map. Note that computing the hash might impose an additional overhead. Also, rebalancing shards is difficult.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Materialized View Pattern

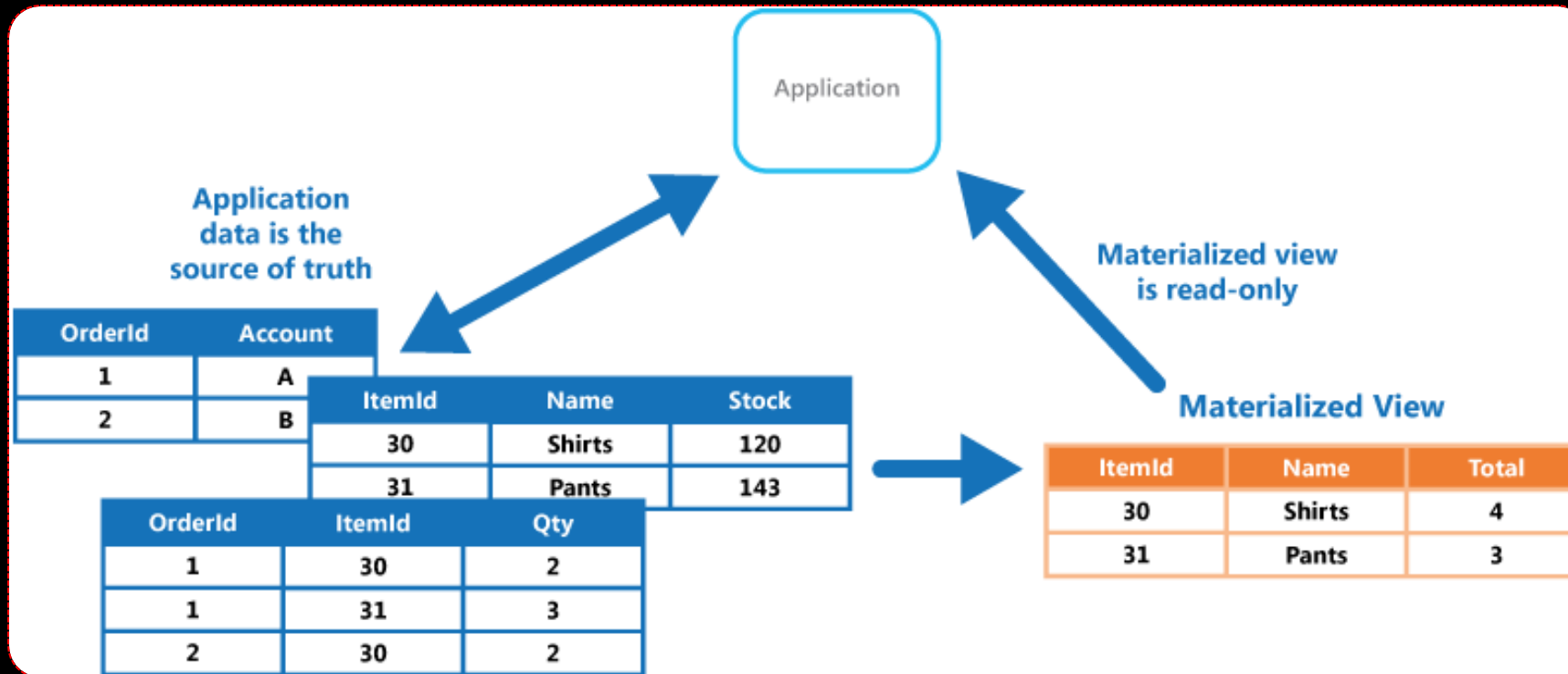
- When storing data, the priority for developers and data administrators is often focused on how the data is stored, as opposed to how it's read. The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. For example, when using NoSQL document store, the data is often represented as a series of aggregates, each containing all of the information for that entity.
- However, this can have a negative effect on queries. When a query only needs a subset of the data from some entities, such as a summary of orders for several customers without all of the order details, it must extract all of the data for the relevant entities in order to obtain the required information.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Materialized View Pattern

Materialized View pattern describes generating prepopulated views of data in environments where the source data isn't in a suitable format for querying, where generating a suitable query is difficult, or where query performance is poor due to the nature of the data or the data store.



DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Materialized View Pattern

This pattern is useful when:

1. Creating materialized views over data that's difficult to query directly, or where queries must be very complex to extract data that's stored in a normalized, semi-structured, or unstructured way.
2. Creating temporary views that can dramatically improve query performance, or can act directly as source views or data transfer objects for the UI, for reporting, or for display.
3. Supporting occasionally connected or disconnected scenarios where connection to the data store isn't always available. The view can be cached locally in this case.
4. Simplifying queries and exposing data for experimentation in a way that doesn't require knowledge of the source data format. For example, by joining different tables in one or more databases, or one or more domains in NoSQL stores, and then formatting the data to fit its eventual use.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Materialized View Pattern

This pattern is useful when:

5. Providing access to specific subsets of the source data that, for security or privacy reasons, shouldn't be generally accessible, open to modification, or fully exposed to users.
6. Bridging different data stores, to take advantage of their individual capabilities. For example, using a cloud store that's efficient for writing as the reference data store, and a relational database that offers good query and read performance to hold the materialized views.
7. When using microservices, you are recommended to keep them loosely coupled, including their data storage. Therefore, materialized views can help you consolidate data from your services. If materialized views are not appropriate in your microservices architecture or specific scenario, please consider having well-defined boundaries that align to domain driven design (DDD) and aggregate their data when requested.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Materialized View Pattern

This pattern isn't useful in the following situations:

- The source data is simple and easy to query.
- The source data changes very quickly, or can be accessed without using a view. In these cases, you should avoid the processing overhead of creating views.
- Consistency is a high priority. The views might not always be fully consistent with the original data.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Scheduler Agent Supervisor Pattern

An application performs tasks that include a number of steps, some of which might invoke remote services or access remote resources. The individual steps might be independent of each other, but they are orchestrated by the application logic that implements the task.

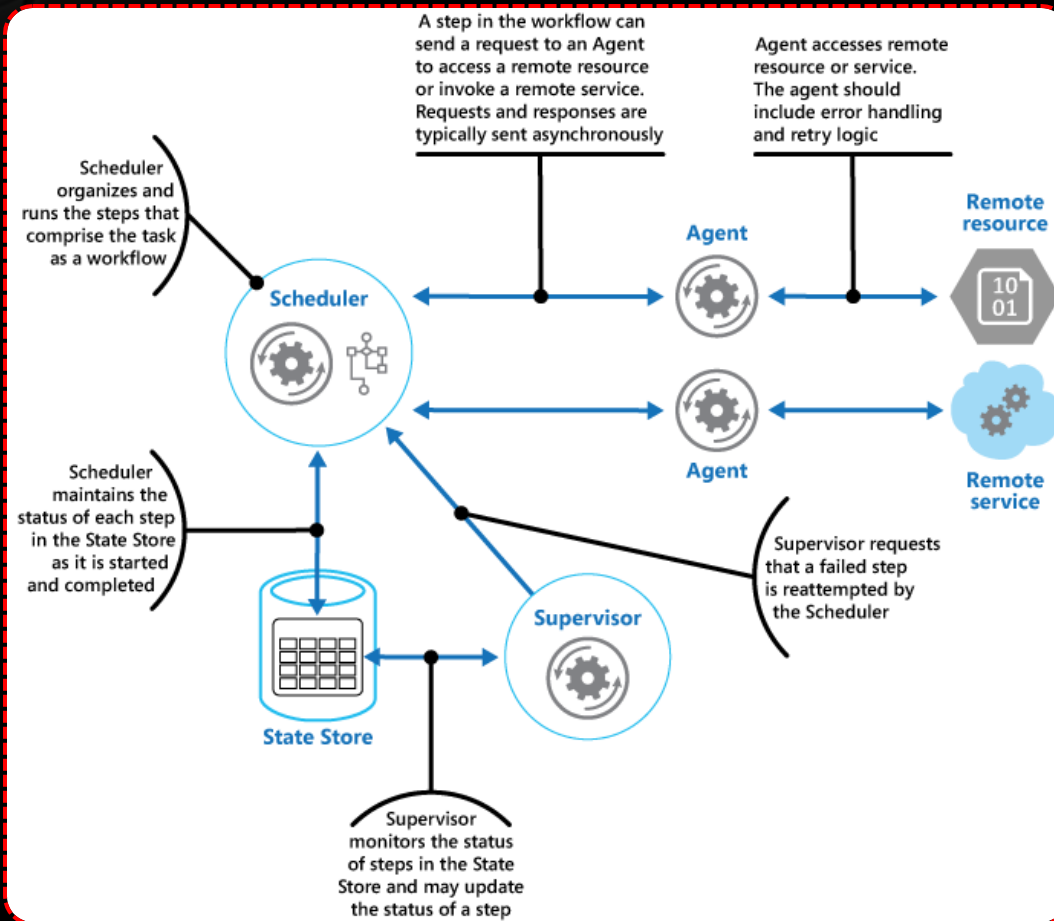
Whenever possible, the application should ensure that the task runs to completion and resolve any failures that might occur when accessing remote services or resources. Failures can occur for many reasons. For example, the network might be down, communications could be interrupted, a remote service might be unresponsive or in an unstable state, or a remote resource might be temporarily inaccessible, perhaps due to resource constraints. In many cases the failures will be transient and can be handled by using the Retry pattern.

If the application detects a more permanent fault it can't easily recover from, it must be able to restore the system to a consistent state and ensure integrity of the entire operation.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Scheduler Agent Supervisor Pattern

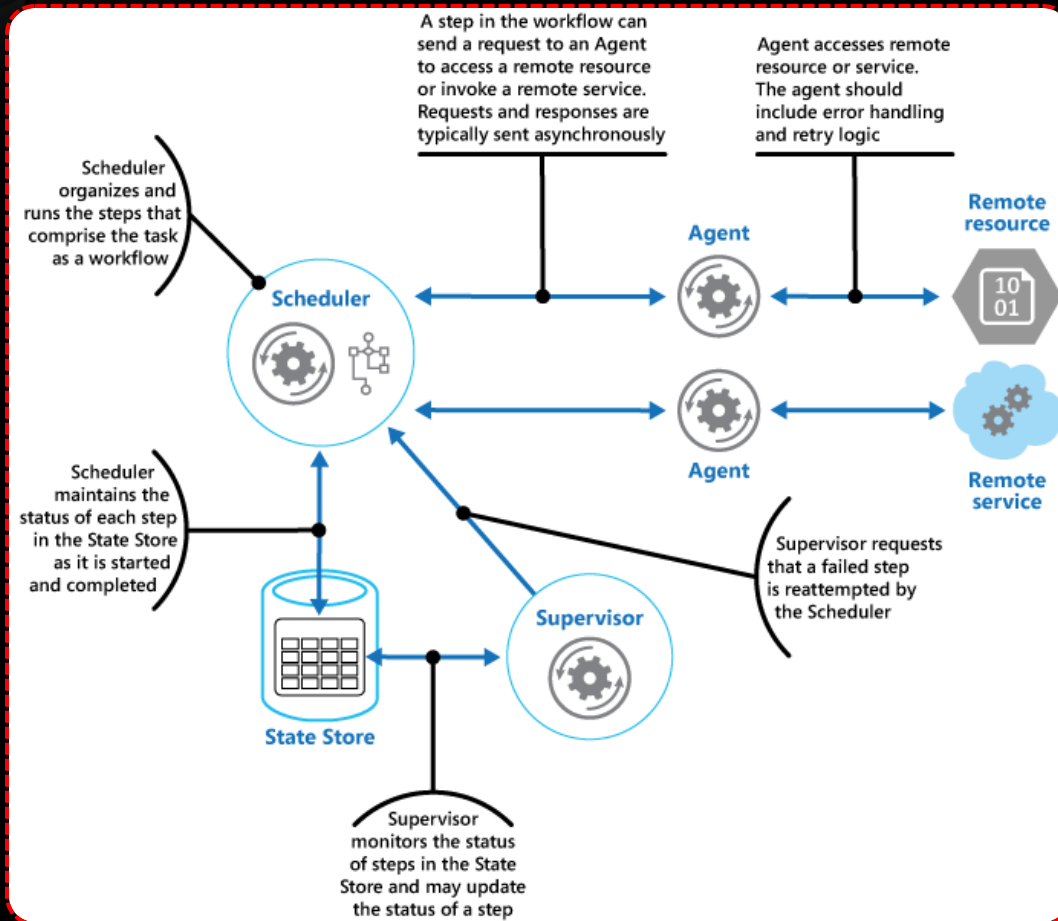


- The Scheduler, Agent, and Supervisor are logical components and their physical implementation depends on the technology being used.
- For example, several logical agents might be implemented as part of a single web service.
- The Scheduler maintains information about the progress of the task and the state of each step in a durable data store, called the state store.
- The Supervisor can use this information to help determine whether a step has failed.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Scheduler Agent Supervisor Pattern

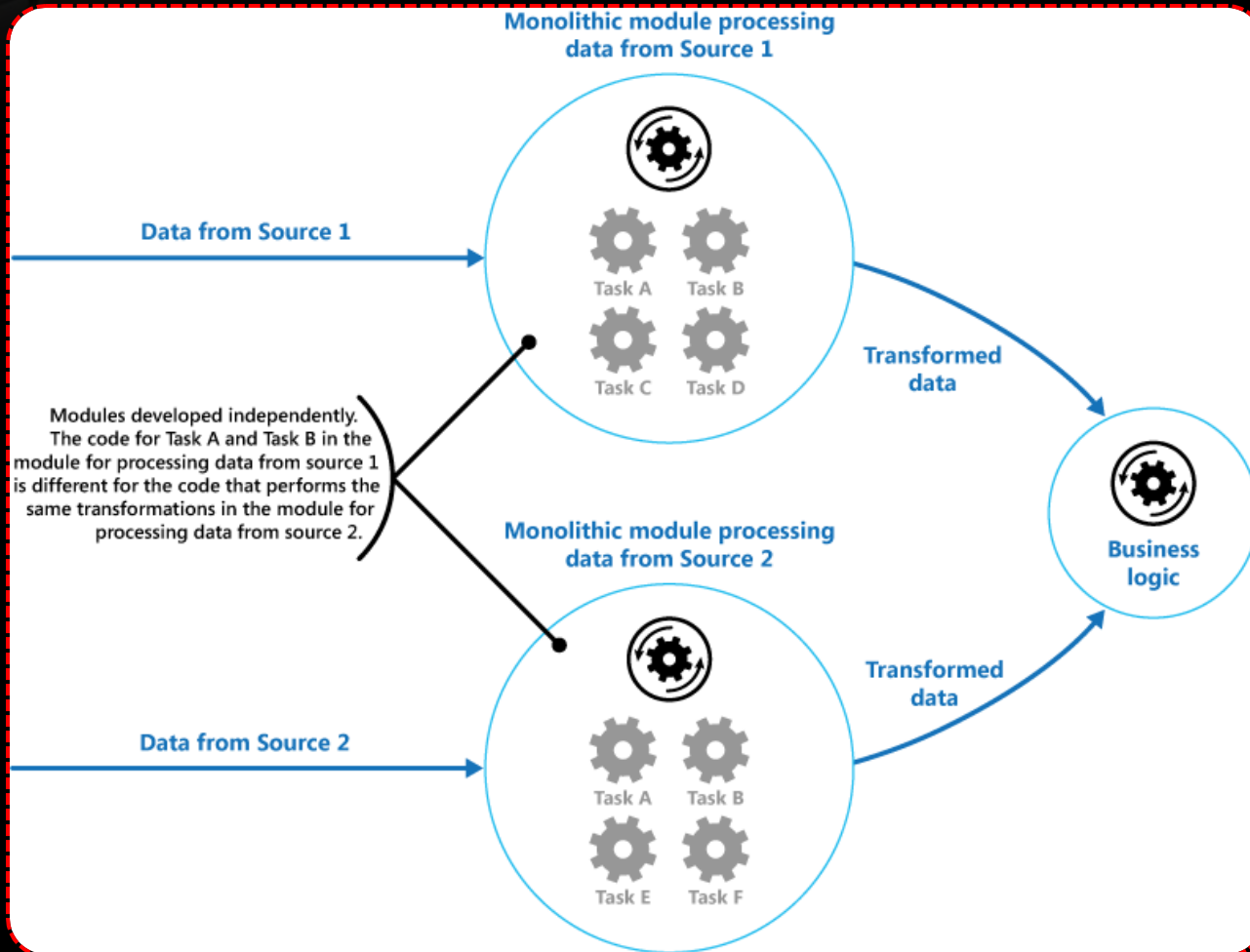


- Use this pattern when a process that runs in a distributed environment, such as the cloud, must be resilient to communications failure and/or operational failure.
- This pattern might not be suitable for tasks that don't invoke remote services or access remote resources.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Pipes and Filters Pattern



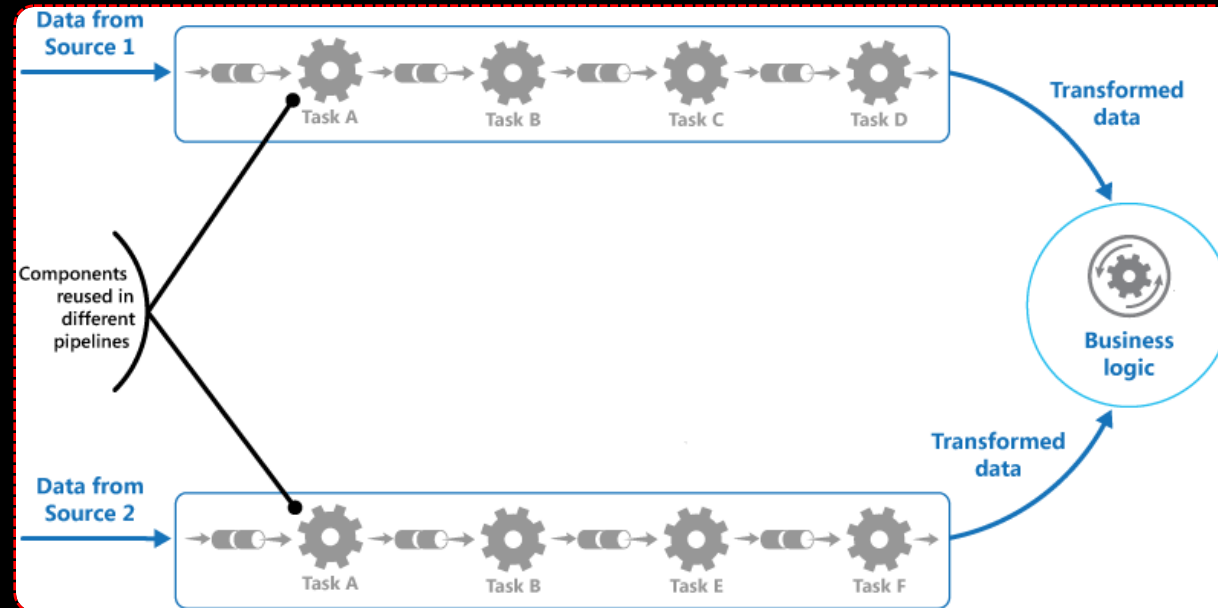
- An application can perform a variety of tasks that vary in complexity on the information it processes.
- However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere in the application.
- The following diagram illustrates the problems with processing data by using the monolithic approach.
- An application receives and processes data from two sources.
- The data from each source is processed by a separate module that performs a series of tasks to transform the data before passing the result to the business logic of the application.

DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Pipes and Filters Pattern

Break down the processing that's required for each stream into a set of separate components (or *filters*), each performing a single task. To achieve a standard format of the data that each component receives and sends, the filters can be combined in the pipeline. Doing so avoids code duplication and makes it easy to remove or replace components, or integrate additional components, if the processing requirements change. This diagram shows a solution that's implemented with pipes and filters:



DESIGN PATTERN OF BIG DATA (STUDY CASE OF AZURE)



Pipes and Filters Pattern

Use this pattern when:

- The processing required by an application can easily be broken down into a set of independent steps.
- The processing steps performed by an application have different scalability requirements.
- You require the flexibility to allow reordering of the processing steps that are performed by an application, or to allow the capability to add and remove steps.
- The system can benefit from distributing the processing for steps across different servers.
- You need a reliable solution that minimizes the effects of failure in a step while data is being processed.

This pattern might not be useful when:

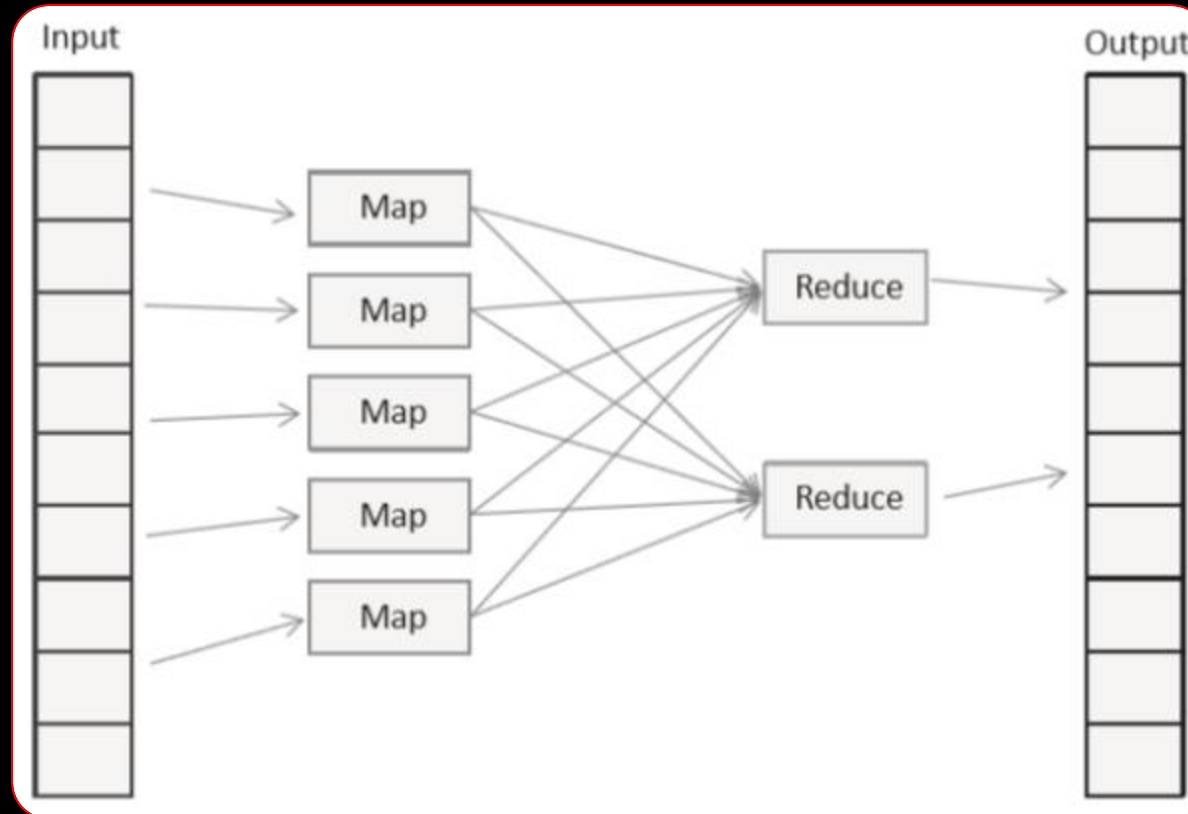
- The processing steps performed by an application aren't independent, or they have to be performed together as part of a single transaction.
- The amount of context or state information that's required by a step makes this approach inefficient. You might be able to persist state information to a database, but don't use this strategy if the extra load on the database causes excessive contention.



POLA MAPREDUCE

POLA MAP-REDUCE

Model MapReduce memiliki dua fase: Map dan Reduce. “**Map**” didefinisikan sebagai proses mapping atau pengelompokan data. Sedangkan “**reduce**” didefinisikan sebagai proses reduksi atau meringkas data.



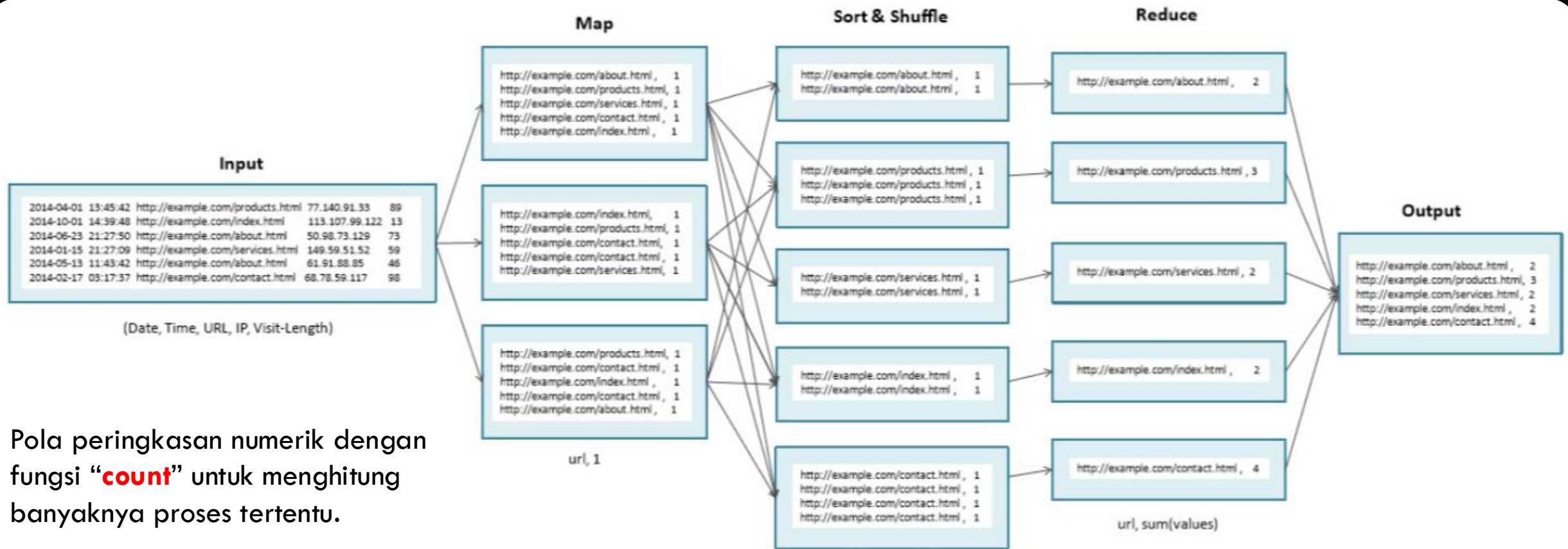
Alur proses MapReduce

POLA MAP-REDUCE



Pola Peringkasan Numerik

Pola peringkasan numerik digunakan untuk menghitung berbagai statistik seperti nilai maksimum, minimum, mean dll.

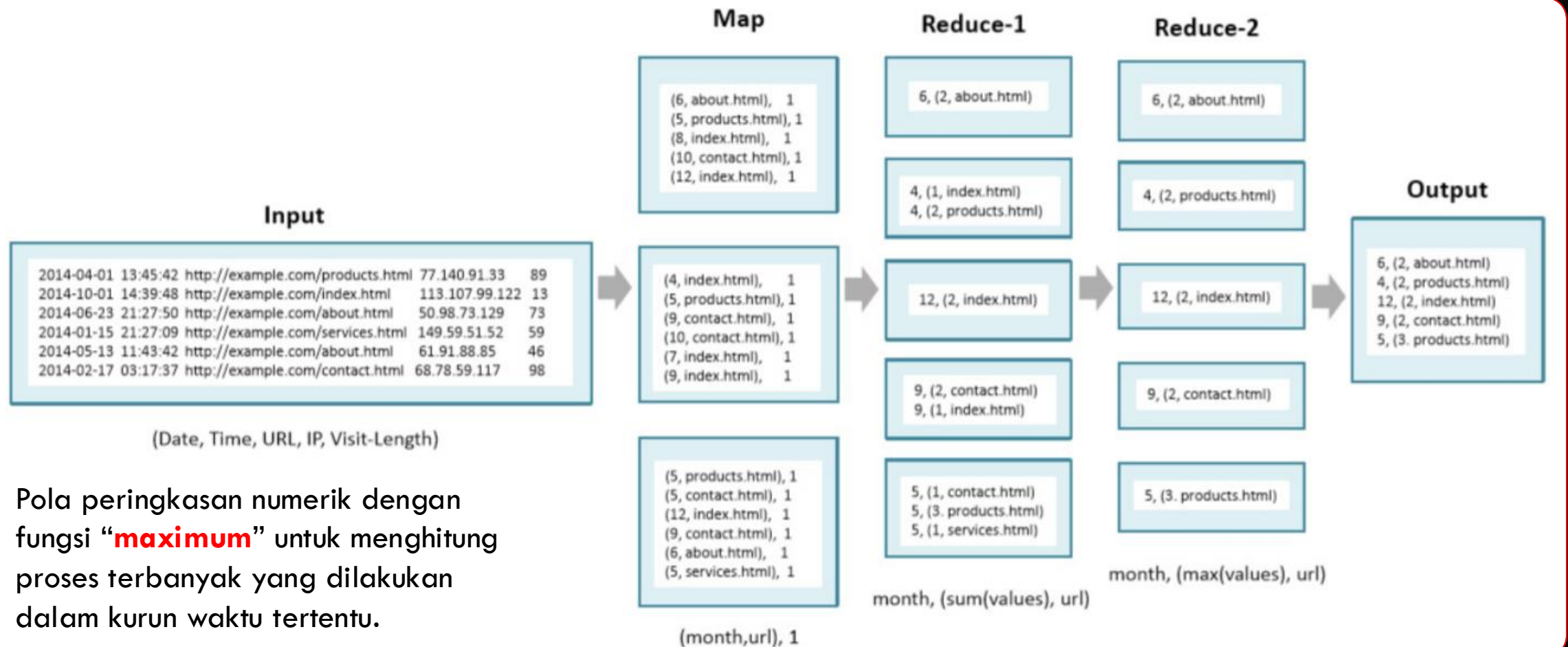


Pola peringkasan numerik dengan fungsi “**count**” untuk menghitung banyaknya proses tertentu.

POLA MAP-REDUCE



Pola Peringkasan Numerik

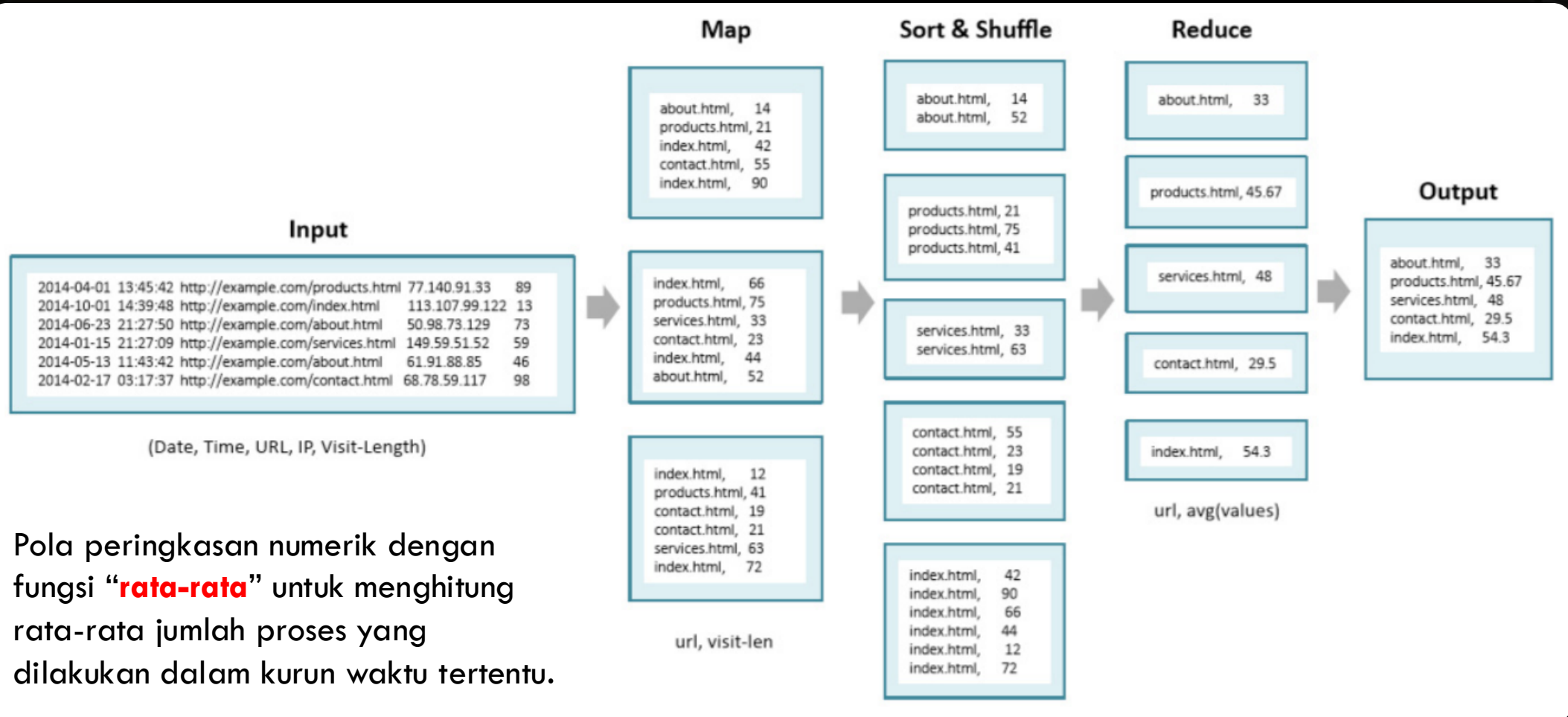


Pola peringkasan numerik dengan fungsi “**maximum**” untuk menghitung proses terbanyak yang dilakukan dalam kurun waktu tertentu.

POLA MAP-REDUCE



Pola Peringkasan Numerik



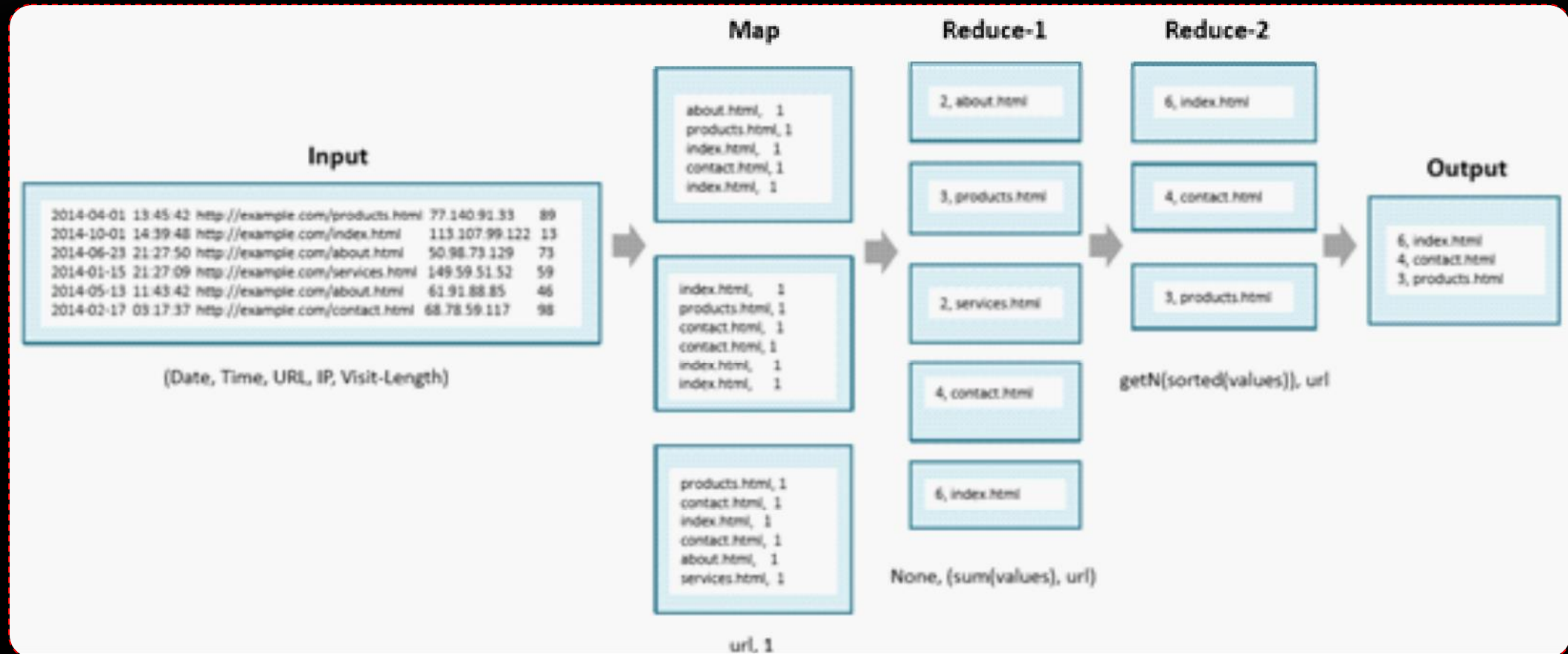
Pola peringkasan numerik dengan fungsi “**rata-rata**” untuk menghitung rata-rata jumlah proses yang dilakukan dalam kurun waktu tertentu.

POLA MAP-REDUCE



Pola Top-N

Digunakan untuk mencari data pada antrian teratas.

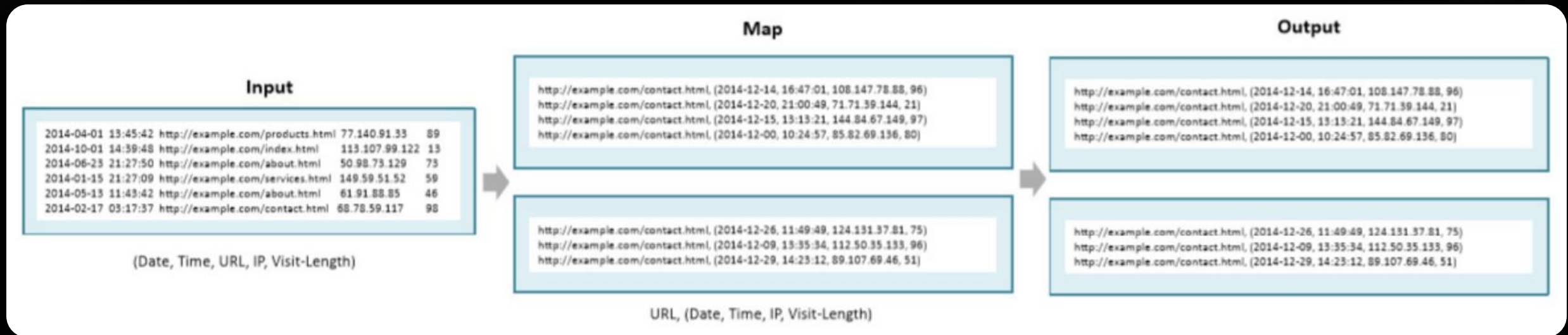


POLA MAP-REDUCE



Pola Filter

Pola filter digunakan untuk menyaring subset rekaman berdasarkan kriteria tertentu.



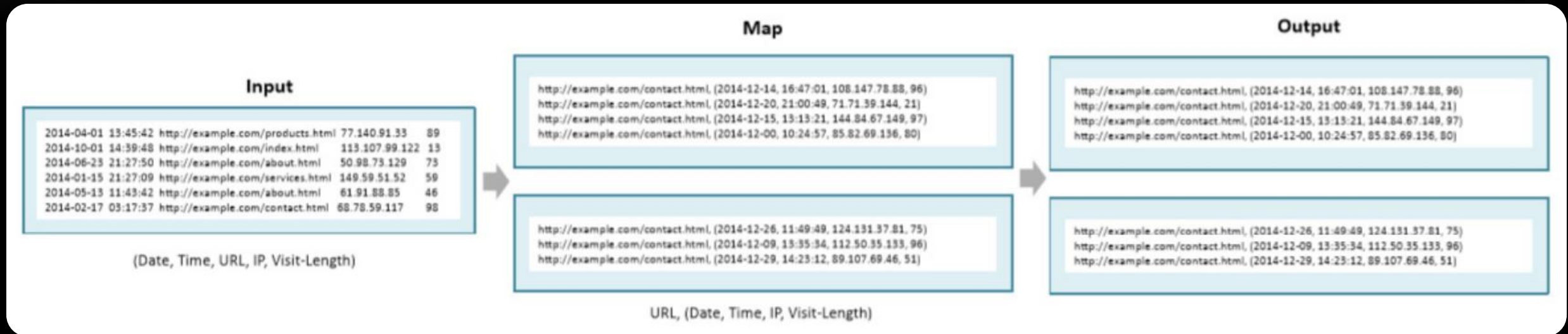
Contoh hasil filter dengan kriteria waktu Desember 2014

POLA MAP-REDUCE



Pola Filter

Pola filter digunakan untuk menyaring subset rekaman berdasarkan kriteria tertentu.



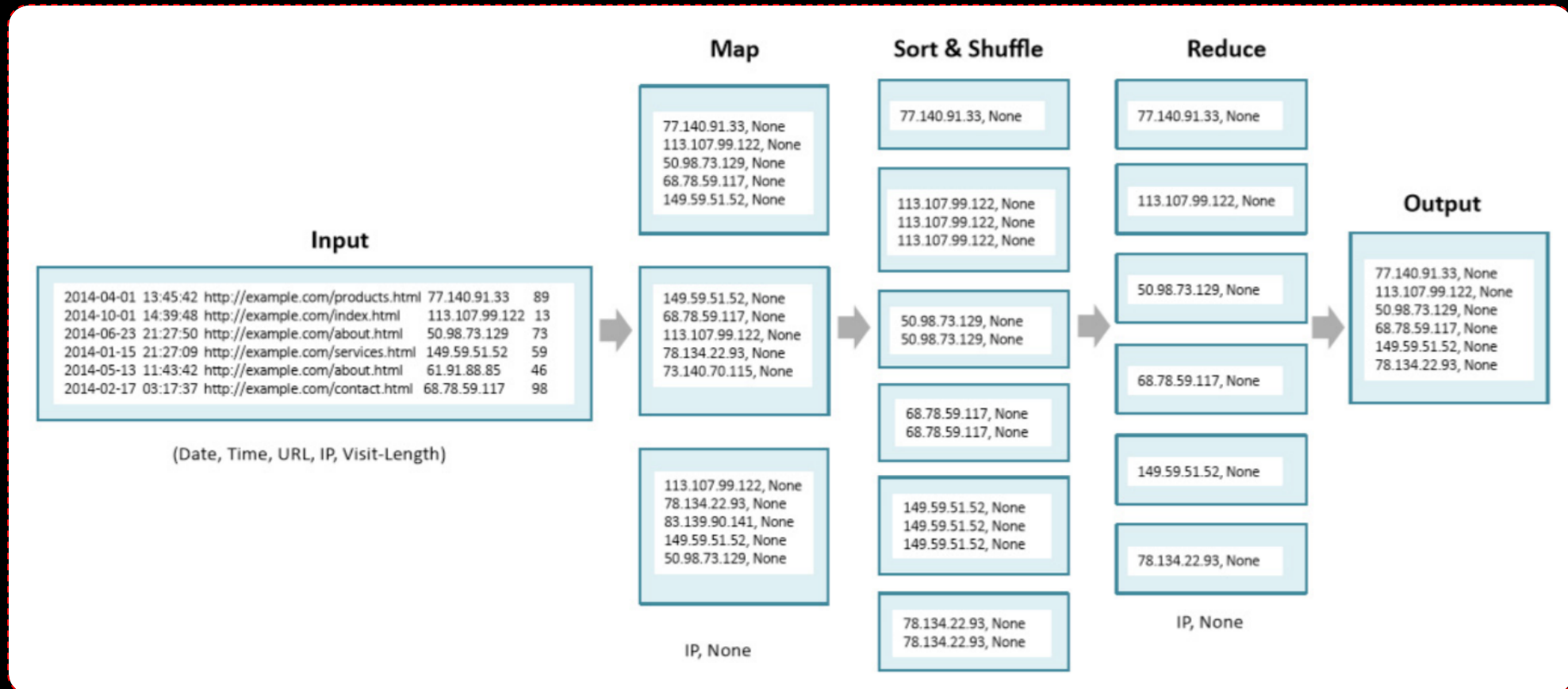
Contoh hasil filter dengan kriteria waktu Desember 2014

POLA MAP-REDUCE



Pola Distinct

Pola Distinct digunakan untuk mencatat dan meringkas alamat sumber produsen yang diakses dalam kurun waktu tertentu.

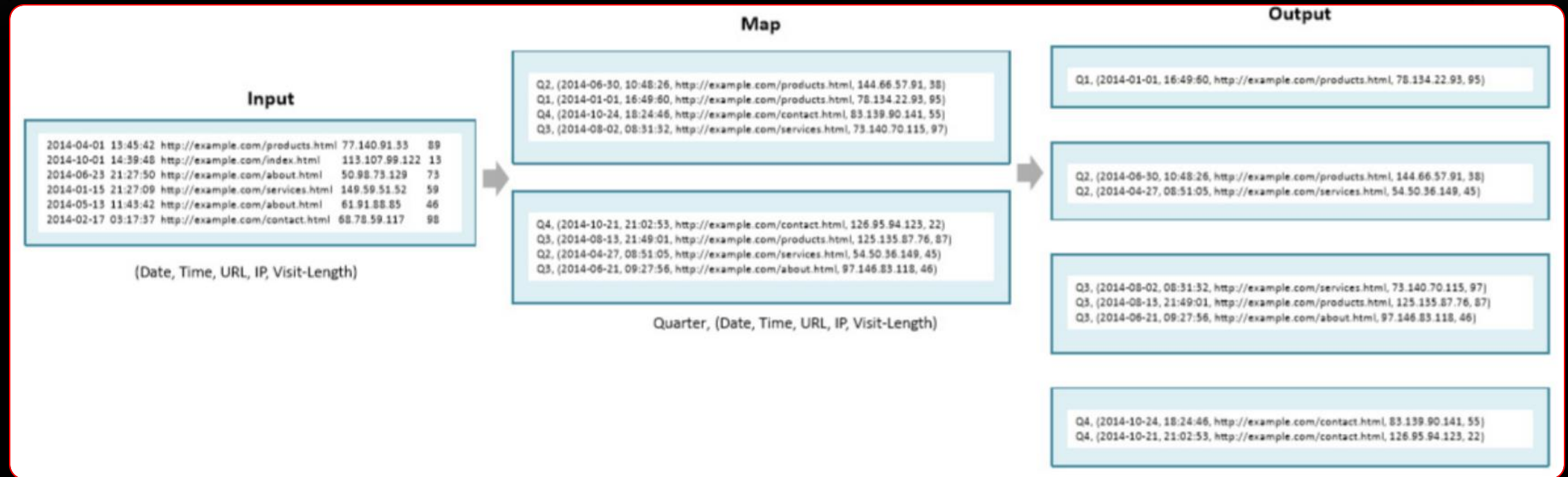


POLA MAP-REDUCE



Pola Binning

Pola Binning digunakan untuk mempartisi record ke dalam bin atau kategori.

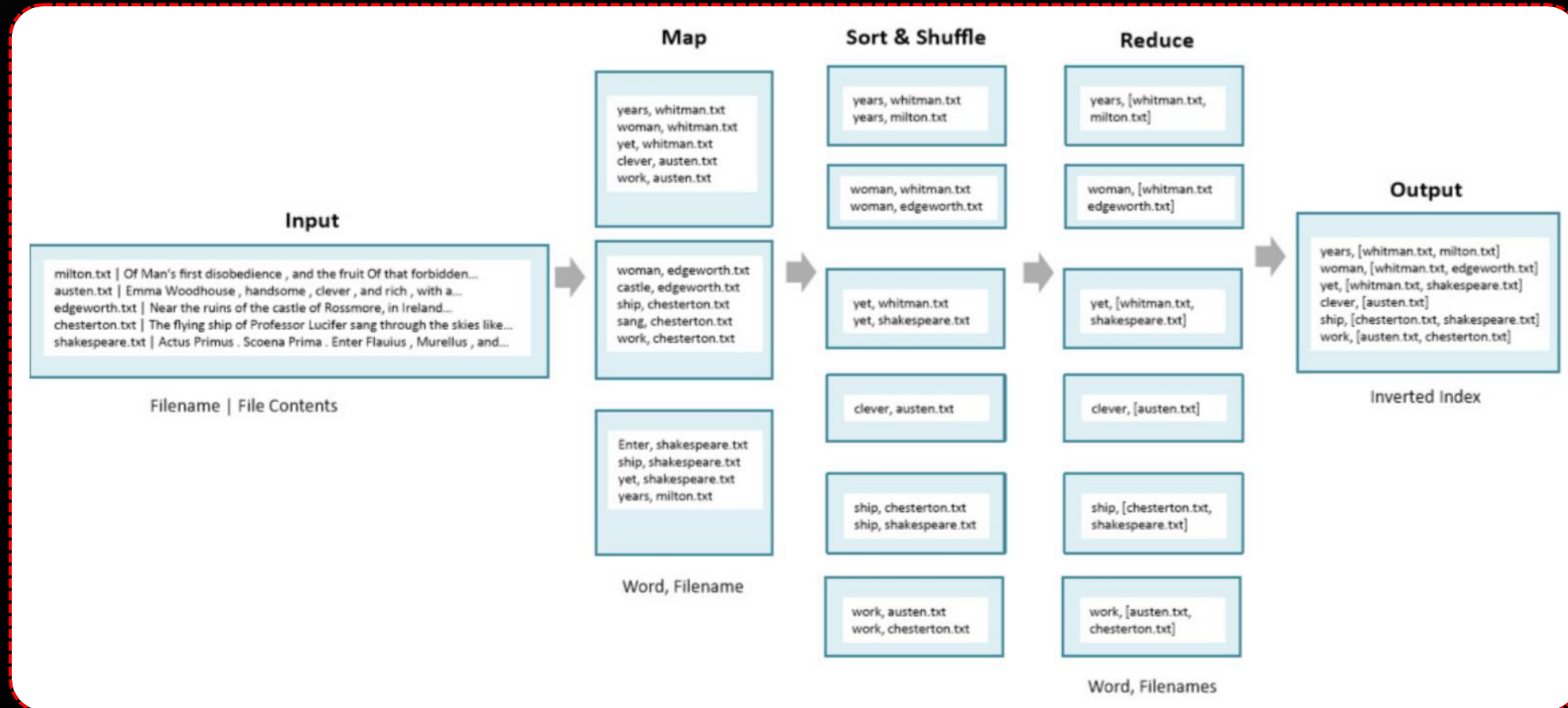


POLA MAP-REDUCE



Pola Inverted Index

Pola Inverted Index digunakan untuk mencatat index setiap proses dan meringkasnya.

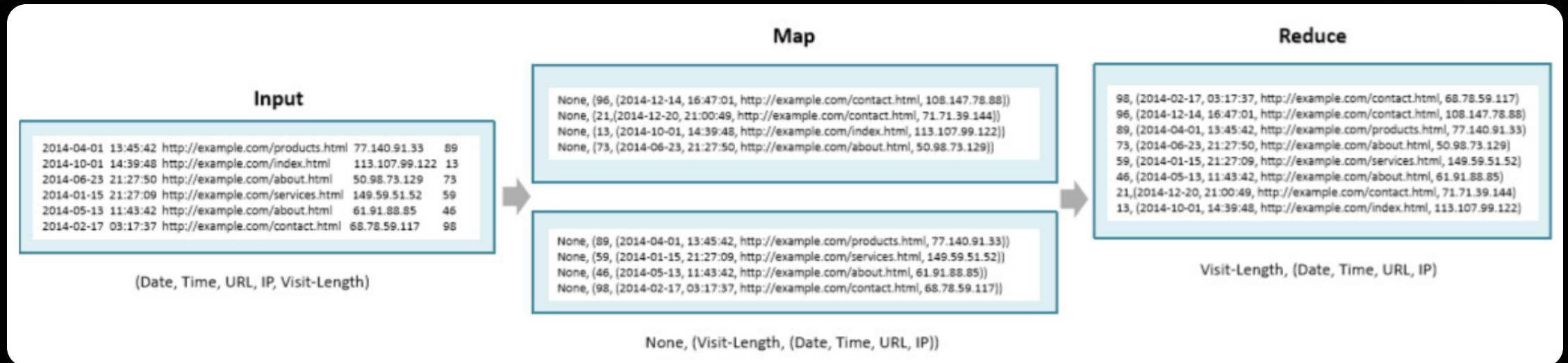


POLA MAP-REDUCE



Pola Sorting

Pola sorting digunakan untuk mengurutkan record berdasarkan field tertentu.

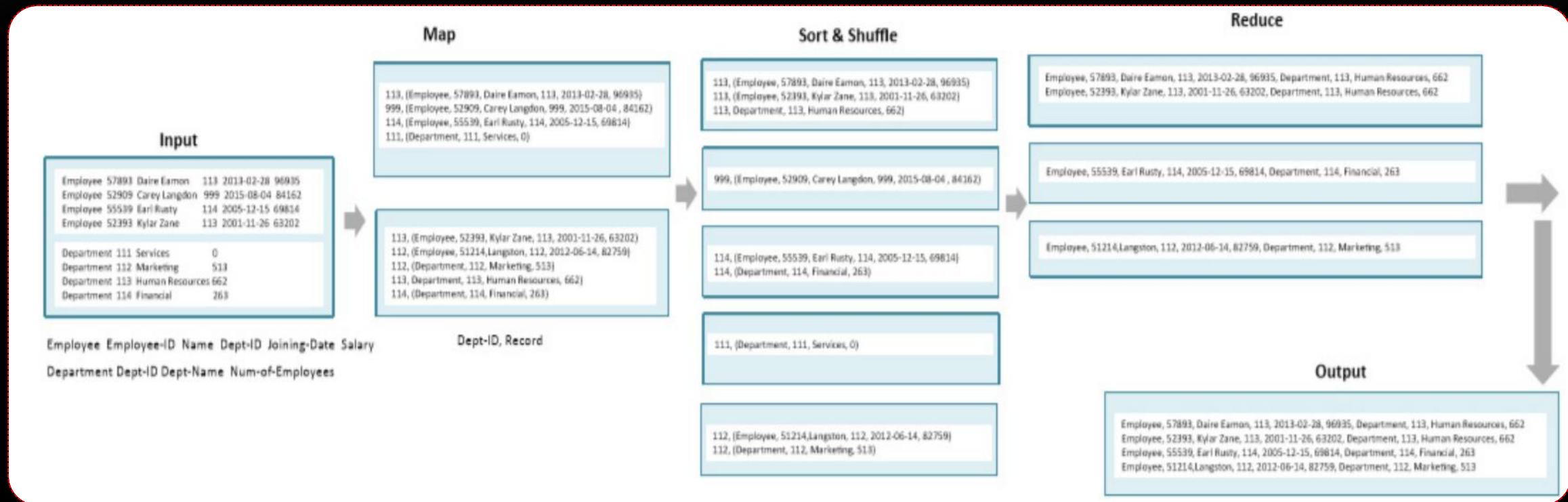


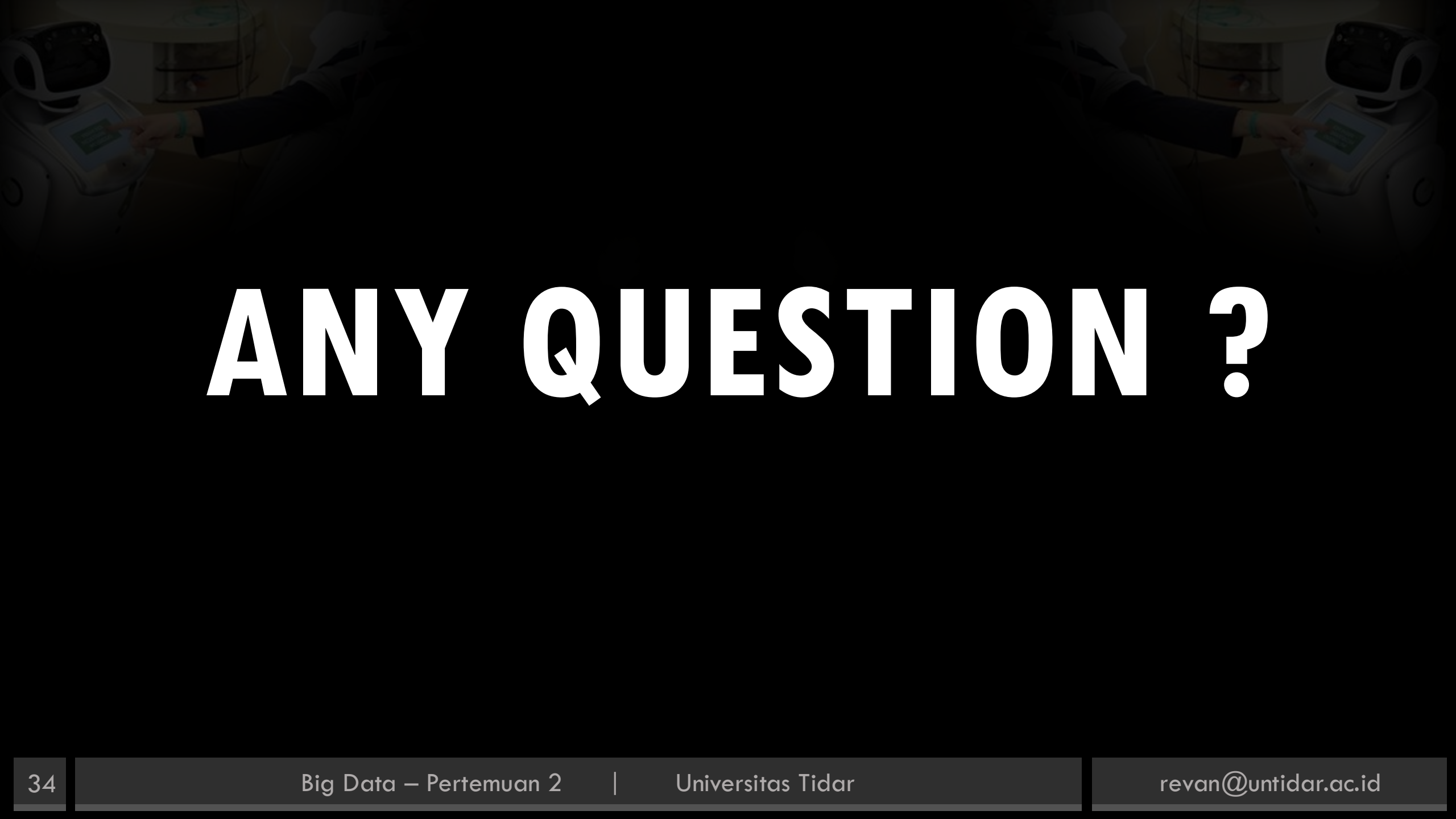
POLA MAP-REDUCE



Pola Join

Pola join digunakan untuk menggabungkan catatan dalam file untuk diproses lebih lanjut.





ANY QUESTION ?