



# Eddies: Continuously Adaptive Query Processing

**Speaker:** Ohoud Alharbi




# Goals of Presentation

- ◎ Consideration for run-time optimization:
  - Synchronization barriers.
  - Moments of Symmetry.
- ◎ What is Eddies?
- ◎ Routing Tuples in Eddies:
  - Naïve scheme
  - Lottery scheme
- ◎ Experiments to evaluate routing schemes



# Steps for a typical Query Processor

- Express query as **algebra expression** (set of operators).
  - Enumerate **alternative plans**.
  - For each alternative plan, **estimate the cost** of each enumerated plan.
  - Choose the plan with the **least estimate cost**.
- 

# Constantly Fluctuating Environment

- ◎ Large scale system that functions an unpredictable and constantly **fluctuating environment**.
- ◎ We need query execution plans to be **reoptimized regularly** during the course of query processing.
- ◎ Allow system to **adapt dynamically** to fluctuation in computing resources, data characteristics and user preference.

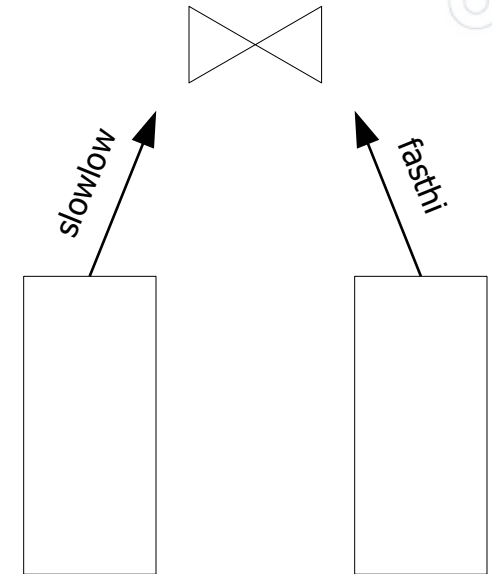
# Constantly Fluctuating Environment

**Run-time reoptimization should consider:**

- **Synchronization barriers:** Where one operation hinders the speed of another operations
- **Moments of Symmetry:** when the query is executed to a point that the optimizer can change the query plan without affecting the way the query plans predicates are performed

# Synchronization Barriers

- Assume you have a **merge join** on two inputs. (slowlo and fasthi)
- The processing of **fasthi** is postponed for a long time while consuming many tuples from **slowlo**.
- Synchronization barriers limit concurrency.
- Desirable to minimize the number of Synchronization barriers.



# Moments of Symmetry

- ◎ You can only **re-optimize** at a **moment of symmetry**.
- ◎ A moment of symmetry is when the query is executed to a point that the optimizer can **change the query plan** without affecting the way the **query plans predicates** are performed

## Example

- ◎ Assume you have a **nested loop** join with inner relation **R** and outer relation **S**.
- ◎ In this example you can only re-optimize this join when **R is completely scanned**.

A decorative background graphic consisting of a network of nodes and edges. The nodes are represented by circles of varying sizes and colors, including light gray, dark gray, and blue. Some nodes are highlighted with a blue outline. The edges are thin gray lines connecting the nodes, forming a complex, interconnected web. The overall style is clean and modern, with a focus on geometric shapes and a limited color palette.

# What is Eddies?

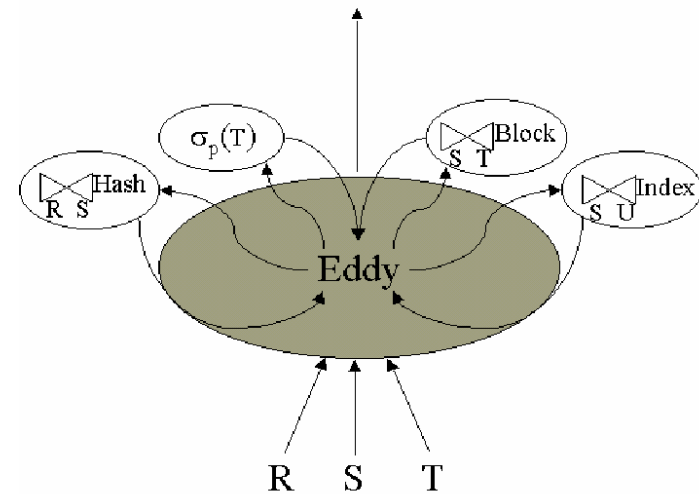
# Eddies

- © Eddies was designed to dynamically **re-optimize queries**.
- © Eddies implemented in a **River**.
- © A River is a shared nothing parallel query processing framework that dynamically adapts to fluctuations and workloads.

# Eddies

- © An eddy is implemented via a module in a river containing an arbitrary number of **input relations**, a number of **binary modules**, and a **single output relation**.
- © Although eddies will reorder tables among joins, a heuristic pre-optimizer must choose how to **initially** pair off relations into joins.

# Eddies



- Continuously reorders the application of pipelined operators in a query plan, on a tuple-by-tuple basis.
- Data flows into the eddy from input relations R, S and T
- The eddy routes tuples to operators: the operators run as independent threads, returning tuples to the eddy
- The eddy sends a tuple to the output only when it has been handled by all the operators
- The eddy adaptively chooses an order to route each tuples through the operators

# Eddies

- ◎ An Eddy also maintains a **fixed size buffer of tuples** that need to be processed.
- ◎ Each operator takes **two tuples**, processes them and delivers them back to the eddy.
- ◎ Each tuple entering eddy has a **tuple descriptor**.
- ◎ A tuple descriptor contains a vector of **Ready bits** and **Done bits**.
- ◎ The Eddy **ships the tuple** to only the operators that have the Ready bits turned on.

# Eddies

- ◎ After an operators is processed it's **done bits are set**
- ◎ If all **done bits are set** the tuple is sent to the **Eddy's output**
- ◎ Eddies preserves the **ordering constraints** while **maximizing opportunities** for tuples to follow different possible ordering of the operators.

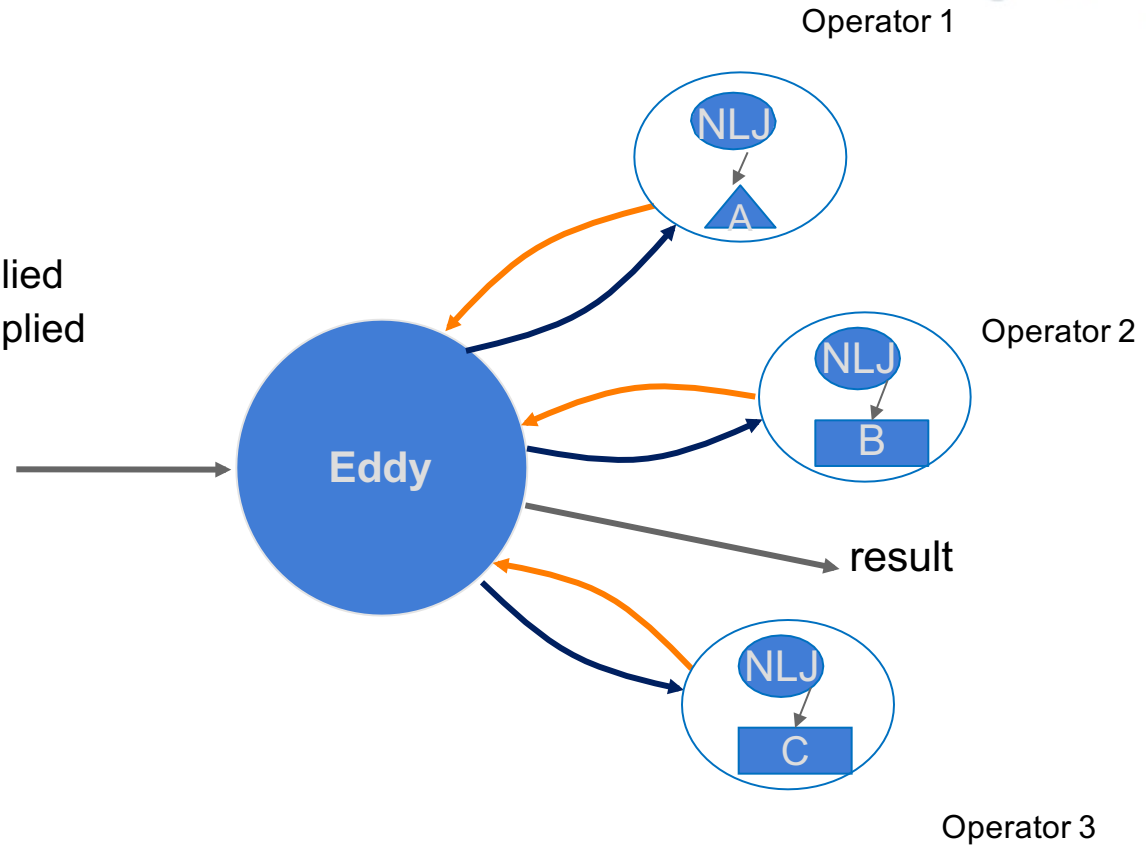
# Eddies Example

ready	done
111	000

ready bit i :

1 : operator i can be applied

0 : operator i can't be applied



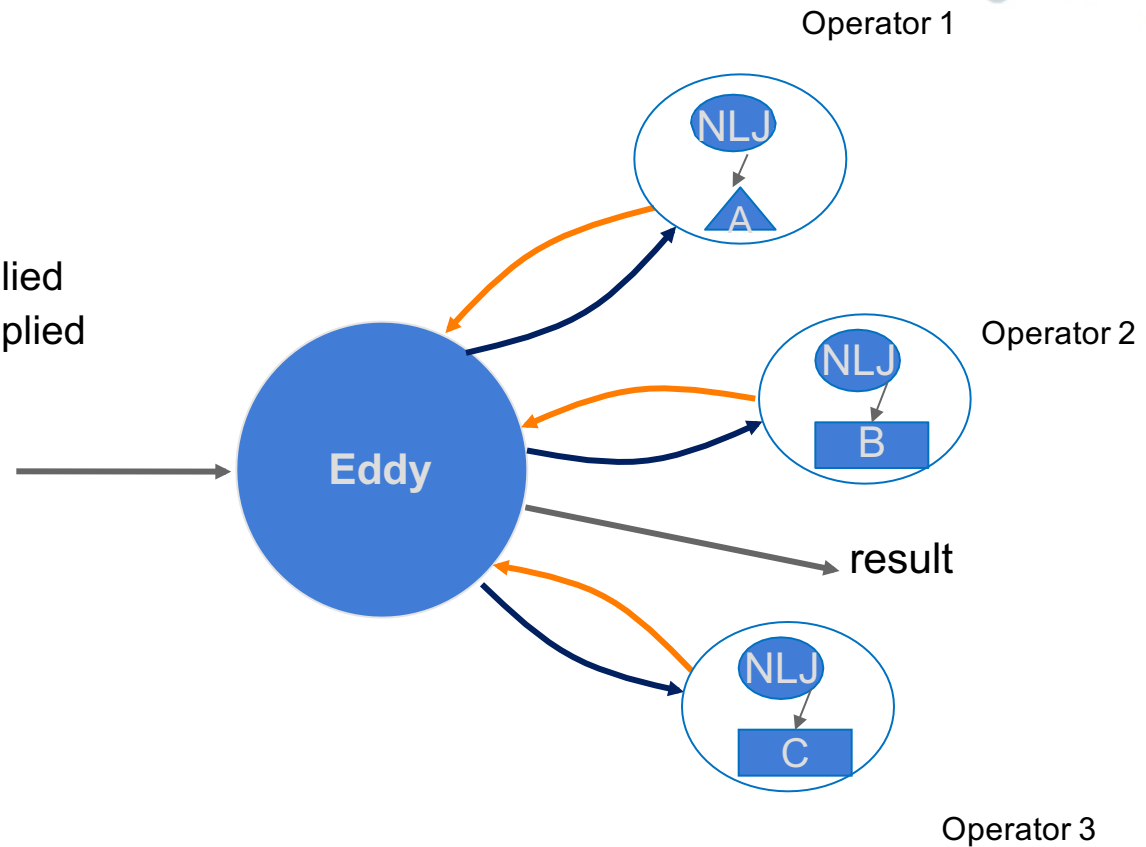
# Eddies Example

ready	done
000	111

ready bit i :

1 : operator i can be applied

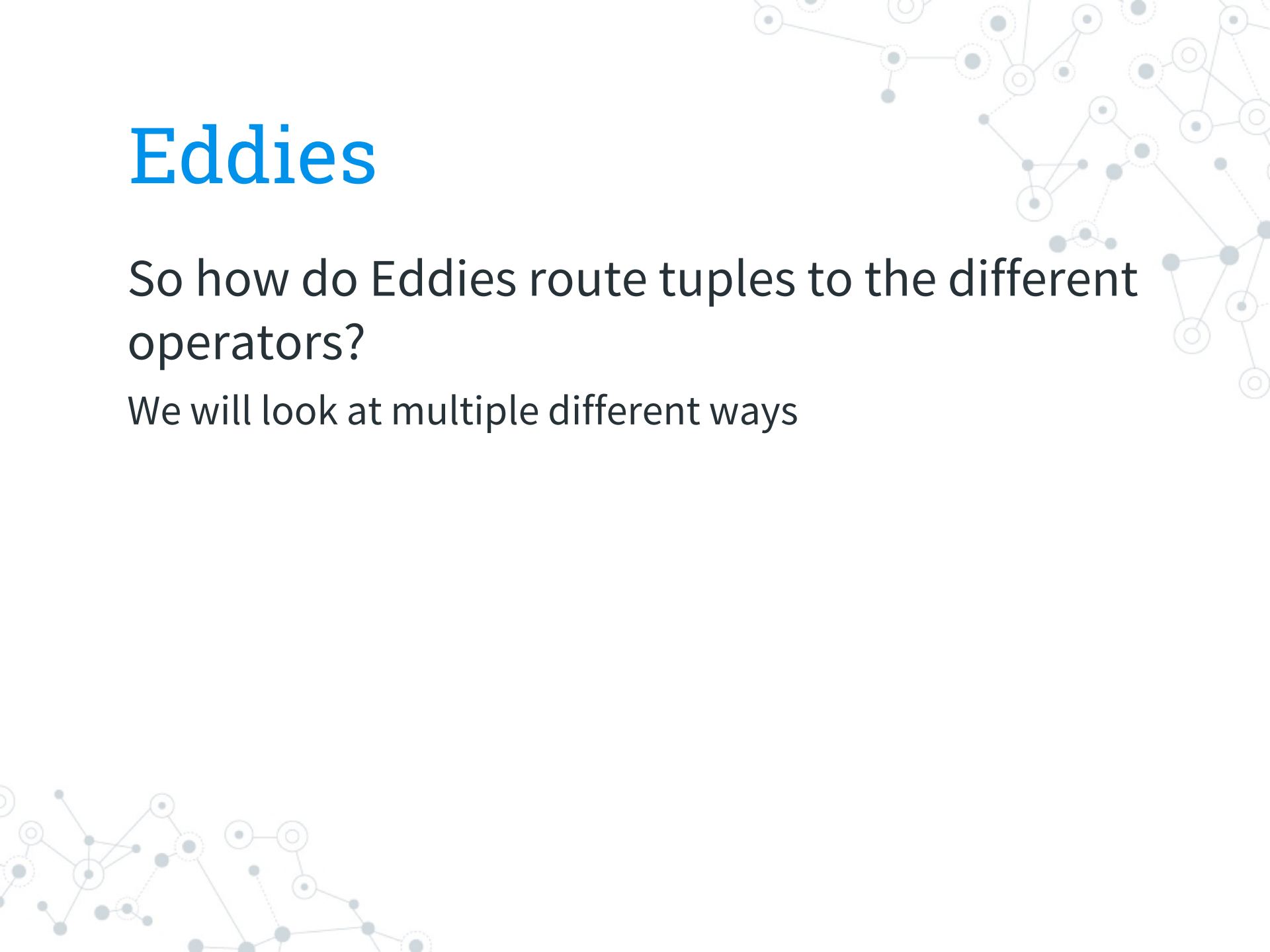
0 : operator i can't be applied



# Eddies

So how do Eddies route tuples to the different operators?

We will look at multiple different ways



A decorative background graphic consisting of a network of nodes and edges. The nodes are represented by circles of varying sizes and colors (gray, blue, and white with blue outlines). The edges are thin gray lines connecting the nodes. The network is more dense on the left and right sides of the slide, with a large cluster on the left and a smaller one on the right. The central area is mostly white, providing space for the title.

# Routing Tuples in Eddies

# Routing Tuples in Eddies

An eddy's tuple buffer is implemented as a **priority queue** with a flexible prioritization scheme.

# Naïve Scheme

- ◎ Eddies buffer is implemented as a **priority queue**
- ◎ When a tuple enters a buffer it's **priority is set to low**
- ◎ After it's **processed by an operator** in the Eddy and returned to the buffer it's **priority is set to high**
- ◎ This ensures that tuples do not get stuck in the Eddy.  
I.e. starvation

# Naïve Scheme

- ◎ This scheme **dynamically adjusts**.
- ◎ Operators that are **slower** (i.e. take 4 seconds vs. 1 second will receive less tuples)
- ◎ Note each operator has a **fixed size queue**.
- ◎ Once **queue is filled** up no more tuples can be inserted into queue.

# Lottery Scheme

- ◎ Need a **learning algorithm** to track both consumption and production over time
- ◎ Each time a tuple is routed to a operator the operator is **credited with a ticket.**
- ◎ When the operator returns a tuple **one ticket is debited.**
- ◎ Operator must use possessed tickets to **win “lottery”** to get new tuples.
- ◎ Tracks how efficiently a operator drains tuples from the system.

# Lottery Scheme

- ◎ Only the operators that have their **Ready bit sets** can participate in the lottery
- ◎ An operator's chance of “**winning the lottery**” and receiving the tuple corresponds to the count of tickets for that operator.
- ◎ **Dynamically adjusts** to selectivity of operators (as well as cost).
- ◎ Therefore more “efficient” operator -> more tickets -> more likely to win lottery -> more likely to get tuples

# Lottery with Window Scheme

- © Problem with lottery: An operator that gained a lot of ticket initially but then **became slow**.
- © In this scheme the lottery scheme is modified such that the lottery only looks at tickets gained by an operator in a **fixed window**.

# Lottery with Window Scheme

## Keeps track of two types of tickets:

- **Banked tickets:** Used when running the lottery.
- **Escrow tickets:** Used to measure efficiency during the window.

## At the end of a window:

Banked Tickets = Escrow Tickets

Escrow Tickets = 0

**Ensure operators re-approve themselves each window**

A decorative network diagram in the top-left corner of the slide. It features a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are solid blue, some are solid grey, and some are hollow with a blue outline. The edges are thin grey lines connecting the nodes. The overall structure is a dense, interconnected mesh.

# Routing Scheme Comparison

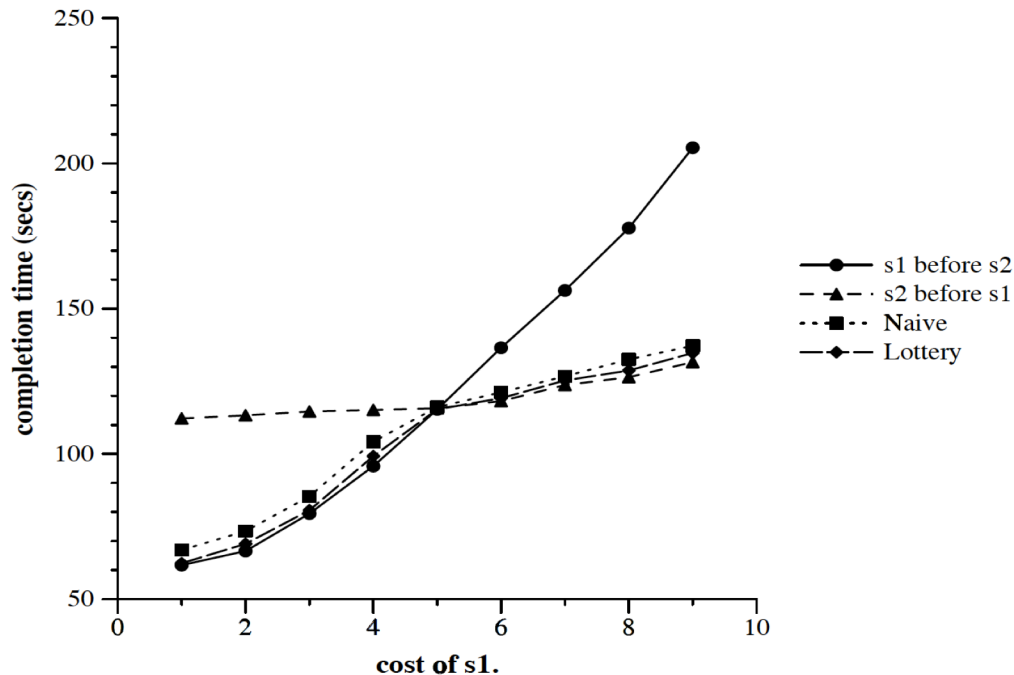
A decorative network diagram in the bottom-right corner of the slide. It features a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are solid blue, some are solid grey, and some are hollow with a blue outline. The edges are thin grey lines connecting the nodes. The overall structure is a dense, interconnected mesh.

# Experiment 1: One Table Query

```
SELECT  *  
FROM    U  
WHERE   s1() AND s2();
```

- ◎ The query above run multiple times, always setting the cost of **s2** = **5** delay units, and
- ◎ In each run they used a different cost for **s1** , varying it between **1 and 9** delay units across runs.
- ◎ The **selectivities** of both selections to **50%**.

# Comparison by Cost



◎ The Naïve approach naturally adjusts based on the cost of operators.

◎ Shows that Lottery also adjusts based on the cost of operators.

# Comparison by Selectivity

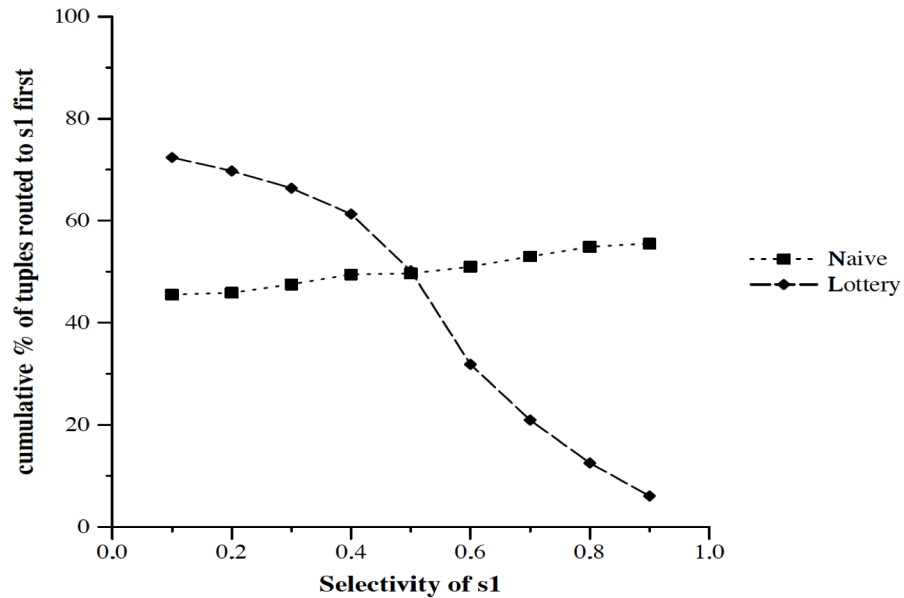


Figure 6: Tuple flow with lottery scheme for the variable-selectivity experiment(Figure 5).

- ⊙ Naïve eddies does not adjust based on selectivity.
- ⊙ Lottery does adjust based on selectivity.

## Experiment 2: 3 tables query

```
SELECT  *  
FROM     $R, S, T$   
WHERE    $R.a = S.a$   
AND      $S.b = T.b$ 
```

## Joins

- Shows that Lottery performs nearly optimally
- Naïve performs between the best and worst case.

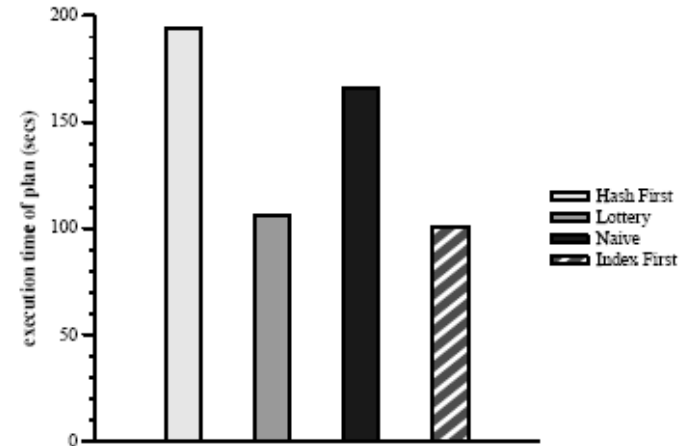



Figure 7: Performance of two joins: a selective Index Join and a Hash Join



# Outline:

- ◎ Consideration for run-time optimization:
    - Synchronization barriers.
    - Moments of Symmetry.
  - ◎ What is an Eddies?
  - ◎ Routing Tuples in Eddies:
    - Naïve scheme
    - Lottery scheme
  - ◎ Experiments to evaluate routing schemes
- 

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. Some nodes are highlighted with blue circles, and others with blue dots.

**Thanks!**

**Any Questions?**

A decorative network diagram in the bottom-right corner, featuring a complex web of interconnected nodes and lines. Some nodes are highlighted with blue circles, and others with blue dots.



# **Eddies: Continuously Adaptive Query Processing**

**Speaker:** Ohoud Alharbi

