Conjunctive Query Processing
[A Formal Model for Theoretical Focus]

Zeyuan Hu
April 28th, 2021
Motivation for the Model

• Given a query on $k$ relations, each of $n$ rows (i.e., a $k$-way join), naively
  • Processing time: $O(n^k)$
  • Size of the output, also, $O(n^k)$

• If basic complexity models are our guide, even simple queries should be infeasible (e.g. $n = 1,000,000$ and $k = 5$)
What happens in practice?

• Joins are often with high reduction factor (i.e., low selectivity)
• Example: \( R \bowtie S \) on the the primary key \( p \) of \( R \)
  • Assume the selectivity for \( p \) is \( \frac{1}{n} \) (i.e., there is 1 output result for each primary key of \( R \))
  • Output size estimation is no longer \( O(n^2) \) but \( O(n) \left( \frac{1}{n} \times n^2 \right) \)
• Relational queries usually work subject to good optimization choices
  • \( \rightarrow \) can still be slow
  • \( \rightarrow \) can be volatile in their performance
Conjunctive Queries (CQ)

• A subset of relational algebra

• Goals of studying CQ
  • Enable theoretical study of the algorithmically hard part of queries
  • Help explain (and thus help resolve) peculiar system behavior
  • Develop new algorithms and *hopefully* impact practice
Full Conjunctive Query

• In Relational Algebra
  • Natural join of \( l \) relations with \( O(n) \) tuples each, no projection
  • \( Q(A_1, A_2, A_3, A_4) = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \)

• In Datalog
  • \( Q(A_1, A_2, A_3, A_4) \leftarrow R_1(A_1, A_2), R_2(A_1, A_2, A_3), R_3(A_2), R_4(A_1, A_2, A_4) \)

• In SQL, full CQ = SELECT ... FROM ... WHERE statement
  • WHERE contains only equalities
  • No projection
Full Conjunctive Query

Other parameters:
- Query size: $O(l)$ (e.g., $l = 4$ for above query)
- Join output result size cardinality: $r$
With Tight Focus on the Computational Challenge

• Main concern: come up algorithms that can evaluate query fast
• Query evaluation problem is known to be NP-Complete
  • No algorithm exists to evaluate any possible query correctly and runs in polynomial time
  • Not a death sentence yet!
  • NP-Complete → algorithm cannot have all three properties
    • General purpose. The algorithm accommodates all possible inputs of the computational problem
    • Correct. For every input, the algorithm correctly solves the problem.
    • Fast. For every input, the algorithm runs in polynomial time.

• Choose one to compromise – General Purpose
A Critical Special Case: **Acyclic** Conjunctive Query

- CQs into fall two classes
  - Acyclic CQ
  - Cyclic CQ
- A **polynomial** algorithm exists to evaluate acyclic CQ
  - Yannakakis Algorithm – a three-pass algorithm
    - $O(\max(r, kn))$ where $r$ is the size of the output, $kn$ is the size of the input
Acyclicity

• A query is acyclic iff it has at least one of these properties
  1. a join tree
  2. a full reducer
  3. An acyclic hypergraph*

* Historically, query acyclicity was independently defined with different notations. They are shown to be equivalent.
Running example

\[ Q(A_1, A_2, A_3, A_4) \leftarrow R_1(A_1, A_2), R_2(A_1, A_2, A_3), R_3(A_2), R_4(A_1, A_2, A_4) \]

• Goal: show \( Q \) is acyclic through three properties above
Property 1: query has a join tree

- Join tree = acyclic query graph + connectedness condition
- query graph – introduced and leveraged for DP-based query opt.
  - Relations are nodes
  - Edges are joins
- Connectedness condition:
  - Def 1: For each attribute $A$, the nodes containing $A$ form a connected subtree
  - Def 2: For each pair of nodes $R$ and $S$ that have common attributes, the following conditions hold:
    - $R$ and $S$ are connected
    - All variables common to $R$ and $S$ occur on the unique path from $R$ to $S
Example

• Suppose we have a database that contains $U(C), N(C, A), E(C, A)$

![Join tree](image)

![Not a Join tree](image)

• A query is acyclic if we can find a join tree
  • can be done in linear time!
Example

- \( Q(A_1, A_2, A_3, A_4) \leftarrow R_1(A_1, A_2), R_2(A_1, A_2, A_3), R_3(A_2), R_4(A_1, A_2, A_4) \) is acyclic because we can find a join tree
Property 2: query has a full reducer

- A full reducer = a semi-join program that remove all dangling tuples in relations
  - Semi-join program = a set of semi-join operations (i.e., semi-join reduction)
  - Dangling tuples = tuples that are not part of final join result

- Example:
  - \( Q(A_1, A_2, A_3, A_4) \leftarrow R_1(A_1, A_2), R_2(A_1, A_2, A_3), R_3(A_2), R_4(A_1, A_2, A_4) \) has a full reducer (and thus acyclic)
    - \( R_2 \Join R_4, R_2 \Join R_3, R_1 \Join R_2, R_2 \Join R_1, R_3 \Join R_2, R_4 \Join R_2 \)
    - Full reducer doesn’t depend on the actual data of each relation!
    - How do you find a full reducer?
Find a full reducer – a two pass process

• \( Q(A_1, A_2, A_3, A_4) \leftarrow R_1(A_1, A_2), R_2(A_1, A_2, A_3), R_3(A_2), R_4(A_1, A_2, A_4) \)

• Suppose we have a join tree of \( Q \), we can construct a full reducer by
  • Semi-join reduction sweep from leaves to root
    • \( R_2 \bowtie R_4, R_2 \bowtie R_3, R_1 \bowtie R_2 \)
    • Semi-join reduction sweep from root to leaves
      • \( R_2 \bowtie R_1, R_3 \bowtie R_2, R_4 \bowtie R_2 \)

• Will this work?
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

Slides of this example are from DATA Lab@Northeastern University
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)

Slides of this example are from DATA Lab@Northeastern University
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)

**Slides of this example are from DATA Lab@Northeastern University**
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)

Slides of this example are from DATA Lab@Northeastern University
Example

\[ Q = R_1(A_1, A_2) \Join R_2(A_1, A_2, A_3) \Join R_3(A_2) \Join R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)
Example

\[ Q = R_1(A_1, A_2) \Join R_2(A_1, A_2, A_3) \Join R_3(A_2) \Join R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)

Slides of this example are from DATA Lab@Northeastern University
Yannakakis Algorithm

• Given acyclic conjunctive query represented by a join tree

• Two Phases
  • Apply a full reducer based on join tree
    • Semi-join reduction sweep from leaves to root
    • Semi-join reduction sweep from root to leaves
  • Use the join tree as the query plan and compute the joins bottom up
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)

Slides of this example are from DATA Lab@Northeastern University
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)
3. Join bottom-up

\[ R_2 = R_3 \bowtie R_2 \]
\[ R_2 = R_4 \bowtie R_2 \]
\[ R_1 = R_1 \bowtie R_2 \]

Slides of this example are from DATA Lab@Northeastern University
Property 3: query has an acyclic hypergraph

- A hypergraph for a natural join
  - Node = attribute in query
  - Hyperedge = relation

- Example 1: Triangle Query
  - \( Q(A, B, C) \leftarrow R(A, B), S(B, C), T(C, A) \)
  - Relation \( R(A, B) \) is represented by the hyperedge \( \{A, B\} \)
  - Relation \( S(B, C) \) is represented by the hyperedge \( \{B, C\} \)
  - This hypergraph is actually a graph, since the hyperedges are each pairs of nodes

- Example 2
  - \( Q(A, B, C, D, E, F) \leftarrow R(A, E, F), S(A, B, C), T(C, D, E), U(A, C, E) \)
Hypergraph construction a legacy of “The Universal Relation” war.

• Universal Relation: A concept where all relation schema would be removed and all data merged into a single table.
  • Plausibility: compute cross products as needed, and fill in implausible combinations with NULLs
  • Potential benefit: Obtain certain optimal properties that might not be achievable without removing certain input from a developer.
Hypergraph definition (cont’)

• To define acyclic hypergraph, we need the notion of an “ear” in a hypergraph

• A hyperedge $H$ is an ear if there is some other hyperedge $G$ in the same hypergraph such that every node of $H$ is either:
  • Found only in $H$, or
  • Also found in $G$

• We shall say that $G$ consumes $H$
Ear in Hypergraph Examples

Hyperedge $H = \{A, E, F\}$ is an ear

- $G = \{A, C, E\}$
- Node $F$ is unique to $H$; it appears in no other hyperedge
- The other two nodes of $H$ ($A$ and $E$) are also members of $G$
- What about $\{A, B, C\}, \{C, D, E\}$?

Find ears in this hypergraph
Check Cyclicity of Hypergraph: GYO Algorithm

• GYO Algorithm = a sequence of ear reductions
• An ear reduction = the elimination of one ear from the hypergraph, along with any nodes that appear only in that ear
• A hypergraph is acyclic = the output of GYO algorithm is empty
  • i.e., all hyperedges can be removed by ear reductions
• Properties
  • An ear, if not eliminated at one step, remains an ear after another ear is eliminated
  • Hyperedge that was not an ear, can become an ear after another hyperedge is eliminated
Example

• \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
• Pick one and eliminate it
• Suppose we pick \{A, E, F\}
Example

• \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
• Pick one and eliminate it
• Suppose we pick \{A, E, F\}
Example

- \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
- Pick one and eliminate it
- Suppose we pick \{A, E, F\}
- Next, we pick \{A, B, C\} and eliminate it
Example

• \( \{A, E, F\}, \{A, B, C\}, \{C, D, E\} \) are ears
• Pick one and eliminate it
• Suppose we pick \( \{A, E, F\} \)
• Next, we pick \( \{A, B, C\} \) and eliminate it
Example

• \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
• Pick one and eliminate it
• Suppose we pick \{A, E, F\}
• Next, we pick \{A, B, C\} and eliminate it
• \{A, C, E\} now becomes an ear and eliminate it
Example

• \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
• Pick one and eliminate it
• Suppose we pick \{A, E, F\}
• Next, we pick \{A, B, C\} and eliminate it
• \{A, C, E\} now becomes an ear and eliminate it
Example

- \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
- Pick one and eliminate it
- Suppose we pick \{A, E, F\}
- Next, we pick \{A, B, C\} and eliminate it
- \{A, C, E\} now becomes an ear and eliminate it
- \{C, D, E\} is the only left ear and eliminate it
Example

- \{A, E, F\}, \{A, B, C\}, \{C, D, E\} are ears
- Pick one and eliminate it
- Suppose we pick \{A, E, F\}
- Next, we pick \{A, B, C\} and eliminate it
- \{A, C, E\} now becomes an ear and eliminate it
- \{C, D, E\} is the only left ear and eliminate it
- Original hypergraph is acyclic
Example 2

• Pick an ear to remove
Example 2

• Pick an ear to remove
• No ear to remove $\rightarrow$ hypergraph is cyclic
Example 3

• $Q(A_1, A_2, A_3, A_4) \leftarrow R_1(A_1, A_2), R_2(A_1, A_2, A_3), R_3(A_2), R_4(A_1, A_2, A_4)$

Sequence of ear reductions

• $\{A_2\}$
• $\{A_1, A_2\}$
• $\{A_1, A_2, A_3\}$
• $\{A_1, A_2, A_4\}$

$Q$ is acyclic
Recap

• We have seen three properties for acyclic query
  1. It has a join tree, or
  2. It has a full reducer, or
  3. Its hypergraph is acyclic

• We see how to construct a full reducer from a join tree

• Question: how to find a join tree for a query, if it exists?
Find a Join Tree

• We can construct a join tree during GYO algorithm. In addition to ear reduction, we follow additional steps:
  • Tree nodes = hyperedges
  • The children of a tree node \( H \) are all those hyperedges consumed by \( H \)

• Example
  • \( R(A, B, C), S(B, F), T(B, C, D), G(C, D, E), H(D, E, G) \)
Join Tree 1

• Start to eliminate \(\{A, B, C\}\)
• Since \(\{B, C, D\}\) consumes \(\{A, B, C\}\), \(\{B, C, D\}\) is the parent of \(\{A, B, C\}\)
Join Tree 1

- Start to eliminate \{A, B, C\}
- Since \{B, C, D\} consumes \{A, B, C\}, \{B, C, D\} is the parent of \{A, B, C\}
- Next, remove \{B, F\}, which is also consumed by \{B, C, D\}
Join Tree 1

• Start to eliminate \(\{A, B, C\}\)
• Since \(\{B, C, D\}\) consumes \(\{A, B, C\}\), \(\{B, C, D\}\) is the parent of \(\{A, B, C\}\)
• Next, remove \(\{B, F\}\), which is also consumed by \(\{B, C, D\}\)
• Remove \(\{B, C, D\}\), which is consumed by \(\{C, D, E\}\)
Join Tree 1

• Start to eliminate \{A, B, C\}
• Since \{B, C, D\} consumes \{A, B, C\}, \{B, C, D\} is the parent of \{A, B, C\}
• Next, remove \{B, F\}, which is also consumed by \{B, C, D\}
• Remove \{B, C, D\}, which is consumed by \{C, D, E\}
• Remove \{D, E, G\}, which is consumed by \{C, D, E\}
Join Tree 2

• Start to eliminate \( \{D, E, G\} \)
• Since \( \{C, D, E\} \) consumes \( \{D, E, G\} \), \( \{C, D, E\} \) is the parent of \( \{D, E, G\} \)
Join Tree 2

• Start to eliminate \( \{D, E, G\} \)
• Since \( \{C, D, E\} \) consumes \( \{D, E, G\} \), \( \{C, D, E\} \) is the parent of \( \{D, E, G\} \)
• Remove \( \{C, D, E\} \), which is consumed by \( \{B, C, D\} \)
Join Tree 2

• Start to eliminate \(\{D, E, G\}\)
• Since \(\{C, D, E\}\) consumes \(\{D, E, G\}\), \(\{C, D, E\}\) is the parent of \(\{D, E, G\}\)
• Remove \(\{C, D, E\}\), which is consumed by \(\{B, C, D\}\)
• Remove \(\{B, C, D\}\), which is consumed by \(\{A, B, C\}\)
Join Tree 2

- Start to eliminate \{D, E, G\}
- Since \{C, D, E\} consumes \{D, E, G\}, \{C, D, E\} is the parent of \{D, E, G\}
- Remove \{C, D, E\}, which is consumed by \{B, C, D\}
- Remove \{B, C, D\}, which is consumed by \{A, B, C\}
- Remove \{A, B, C\} and \{B, F\} sequentially
Complexity Notation

• Standard $O$ and $\Omega$ notation for time and memory complexity in the RAM model of computation

• Use $\tilde{O}$-notation (soft-$O$)
  • Abstracts away polylog factors in input size that clutter formulas
  • $O(n^{f(l)} + (\log n)^{f(l)} \cdot r)$ becomes $\tilde{O}(n^{f(l)} + r)$
Data Complexity

• Complexity in query grows in two dimensions:
  • size of query (i.e., number of relations in a multi-way join query)
  • database size (i.e., number of rows contained in each relation of the query)

• Data complexity: the query is fixed (i.e., the size of the query expression itself \( l \) as a constant), and the complexity is expressed in terms of the size of database

• Suppose the query \( Q \) size \( |Q| \) is \( l \), then \( O(f(l) \cdot n^{f(l)} + (\log n)^{f(l)} \cdot r) \) with \( f() \) denote some arbitrary computable function can be simplified to \( O(n^{f(l)} + (\log n)^{f(l)} \cdot r) \)
Lower Bound for Any Join Algorithm

• Join output result size cardinality: \( r \)
• Query size \( l \) (i.e., number of relations in join query)
• \( \Omega(n + r) \) data complexity to compute any query
  • The join algorithm has to read entire input at least once \( \Omega(ln) \)
    (data complexity: \( \Omega(n) \))
  • The join algorithm has to output result \( \Omega(lr) \) (data complexity: \( \Omega(r) \))
    • This the cost of concatenating tuples from \( l \) relations to form the final join result set
• Yannakakis algorithm amazingly matches the lower bound for acyclic CQs with data complexity \( \tilde{O}(n + r) \)
Yannakakis Algorithm

• Given acyclic conjunctive query represented by a join tree
• Two Phases
  • Apply a full reducer based on join tree
    • Semi-join reduction sweep from leaves to root
    • Semi-join reduction sweep from root to leaves
  • Use the join tree as the query plan and compute the joins bottom up
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)
Example

\[ Q = R_1(A_1, A_2) \bowtie R_2(A_1, A_2, A_3) \bowtie R_3(A_2) \bowtie R_4(A_1, A_2, A_4) \]

1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)
3. Join bottom-up

\[ R_2 = R_3 \bowtie R_2 \]
\[ R_2 = R_4 \bowtie R_2 \]
\[ R_1 = R_1 \bowtie R_2 \]

Slides of this example are from DATA Lab@Northeastern University
Yannakakis Algorithm Property

• Key Property
  • No intermediate join result size can be larger than the final result size
  • i.e., each join step can never shrink intermediate result size

• Why?
  • Semi-join reduction removes dangling tuples between pair-wise relations
  • Is it sufficient? No!
  • We need connectedness condition from join tree to ensure all dangling tuples are removed by semi-join reductions
Importance of *connectedness condition*

- Suppose we have a database instance of 
  \{N(“Navy”, 13), U(“Navy”), E(“Navy”, 17)\}
- Final join result: $\emptyset$

\[
N \leftarrow U, N \leftarrow E, U \leftarrow N, E \leftarrow N \\
U = \emptyset, N = \emptyset, E = \emptyset
\]

\[
N \leftarrow U, E \leftarrow U, U \leftarrow N, E \leftarrow E \\
U = \{ (“Navy”) \}, N = \{ (“Navy”, 13) \}, E = \{ (“Navy”, 17) \}
\]
Yannakakis Algorithm Complexity

• Semi-join sweeps take $\tilde{O}(n)$
  • Recall $R \bowtie S = \pi_{\text{attr}(R)}(R \bowtie S)$
  • With sort-merge join, we can compute $R \bowtie S$ in $O(n \log n) = \tilde{O}(n)$
  • There are $2l - 2$ pair-wise semi-join operation, $\tilde{O}((2l - 2)n) = \tilde{O}(n)$ in data complexity

• All intermediate results are of size $O(r)$ b/c key property

• Each join step has $O(n + r)$ input and $O(r)$ output, which can be computed in $\tilde{O}(n + r)$ by sort-merge join ($l$ join steps but ignored in data complexity)

• In total, Yannakakis Algorithm takes $\tilde{O}(n + r)$
Worst-Case Optimal Join Algorithm

Zeyuan Hu
May 3\textsuperscript{rd}, 2021
Recap

• Three properties for acyclic query
  1. It has a join tree, or
  2. It has a full reducer, or
  3. Its hypergraph is acyclic

• How to construct a full reducer from a join tree
• Modify GYO algorithm to construct join tree
• Yannakakis algorithm can run in $\tilde{O}(n + r)$ for acyclic CQ
CQs with Cycles

- 3-path: $Q_{3p} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$
- 3-cycle: $Q_{3c} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_1)$

Slides of this example are from DATA Lab@Northeastern University
What’s Wrong with Cyclic CQ

• Essentially, we cannot find an acyclic query graph that meets *connectedness condition*
  • $\rightarrow$ intermediate results size can be larger than the final result size
  • $\rightarrow$ key property of Yannakakis Algorithm falls through

• Example
  • 3-path: $Q_{3p} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$
  • 3-cycle: $Q_{3c} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_1)$
What’s Wrong with Cyclic CQ (cont’)

• 3-path: $Q_{3p} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$

• 3-cycle: $Q_{3c} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_1)$

• Already semi-join-reduced input

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$A_2$</th>
<th>$A_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$A_3$</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>
What’s Wrong with Cyclic CQ (cont’)

• 3-path: $Q_{3p} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$
• 3-cycle: $Q_{3c} = R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_1)$
• Already semi-join-reduced input
• $R_1 \bowtie R_2$ produces $n^2$ intermediate results
  • Final output size: $n^2$ for $Q_{3p}$, but only $n$ for $Q_{3c}$

---

Slides of this example are from DATA Lab@Northeastern University
What’s Wrong with Cyclic CQ (cont’)

• Both queries have acyclic query graph
• In the right tree, $A_1$ violates connectedness condition

$$Q_{3p} \quad R_3(A_3, A_4) \quad Q_{3c} \quad R_3(A_3, A_1)$$

$$R_2(A_2, A_3) \quad R_2(A_2, A_3)$$

$$R_1(A_1, A_2) \quad R_1(A_1, A_2)$$

• $Q_{3p}$’s query graph is a join tree
Solutions for Cyclic CQ?

• Maybe we just need an algorithm that targets at Cyclic CQ?
• A result that is from ’18 by Ngo et al shows that $\tilde{O}(n + r)$ is unattainable for full CQ based on well-accepted complexity-theoretic assumptions (e.g., P $\neq$ NP)
What Can Be Done?

• Two main ideas
  • Worst-case Optimal Join Algorithms (WCOJA)
  • Tree decompositions

• Tree decompositions
  • Break down a cyclic CQ into query fragments called “bags”
  • Evaluate each query fragment using WCOJA and materialize the result
  • Connect bag results as a join tree and evaluate the whole query using Yannakakis algorithm

• We will focus on WCOJA
Query evaluation problem is known to be NP-Complete
• No algorithm exists to evaluate any possible query correctly and runs in polynomial time
• Not a death sentence yet!
• NP-Complete → algorithm cannot have all three properties
  • General purpose. The algorithm accommodates all possible inputs of the computational problem
  • Correct. For every input, the algorithm correctly solves the problem.
  • Fast. For every input, the algorithm runs in polynomial time.
• Choose one to compromise – General Purpose → Yannakakis Algorithm
• WCOJA chooses different to compromise - Fast
Query Evaluation Problem

• Given
  • a full CQ of the form \( q = R_1(A_1) \bowtie R_2(A_2) \bowtie \ldots \bowtie R_m(A_m) \) where \( A_j \) is the attribute set of relation \( R_j, j \in [m] \)
  • a database instance \( I \) on the schema \( \{ R_1, \ldots, R_m \} \)
• Query evaluation problem is to compute \( q(I) \)
  • \( q(I) = \) a set of tuples \( t \) over attribute set \( \bigcup_{j \in [m]} A_j \) s.t. projection of \( t \) onto the attributes \( A_j \) belongs to \( R_j \), for each \( j \in [m] \)
• Join output result size cardinality: \( r \)
  • \( r \) is database instance dependent
• Yannakakis Algorithm reaches \( \tilde{O}(n + r) \)
Optimal Worst-case Join Evaluation Problem

• An easier problem than query evaluation problem
• Instead of $\tilde{O}(n + r)$, hope to find a polynomial algorithm that can run $\tilde{O}(n + r_{WC})$
  • $r_{WC}$ = maximum possibly output size for the given size of the relations in $q$
• Let $\overline{N} = \{N_1, ..., N_m\}$ and let $I(\overline{N})$ be the set of database instances with $|R_j^I| = N_j$ for $j \in [m]$. Then, $r_{WC} = \sup_{I \in I(\overline{N})} |q(I)|$
  • i.e., supremum (maximum) of all possible $r$ over $I(\overline{N})$
• Even database instance has the same size, the distribution of data can be different and thus we can get different join output size
AGM Bound

• Example:
  • $Q(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$

• How large is $r_{WC}$?
  • Given the sizes of $|R|$, $|S|$, and $|T|$, what is the largest possible query result size $r$?

• Solved by Aterias, Grohe, and Marx in ‘08

• We’ll introduce intuition here
AGM Bound Intuition

• Given $Q(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$ and $|R| = |S| = |T| = N$, what is the bound on the query result size?
• One bound is $O(N^3)$ because we have three-way join and each tuple can be part of final join result. Thus, we have a cartesian product.
• Can we do better? Yes! $O(N^2)$
• Observe that join of any two relations is an upper bound on $\rho$
  • Because we have a triangle query, third relation imposes additional constraint on intermediate relation, which can at best not eliminate any tuples from intermediate relation.
  • $R(a, b) \bowtie S(b, c)$ already gives tuples with attributes $(a, b, c)$, introduce $T$ can remove tuples
AGM Bound Intuition (cont’)

• For \( Q(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c) \), AGM bound gives \( O(N^{1.5}) \)

• How? By generalizing the observation we have for \( Q \) using fractional edge cover

• Edge cover: a set of edges s.t. each vertex in graph \( G \) is an end of at least one edge

• AGM formulate a linear programming problem based on edge cover of hypergraph of \( Q \). Solving such problem leads to the bound.
WCOJA (under graph model)

• We’ll describe WCOJA in the context of graph model using graph pattern matching query (i.e., subgraph query)

• A match is a mapping from variables to constants such that when the mapping is applied to the given pattern, the result is, roughly speaking, contained within the original graph (i.e., subgraph).

• Focus on triangle query
  • $Q(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$
  • In Cypher syntax
    • match (a)-[:TO]->(b)-[:TO]->(c)-[:TO]->(a) return distinct a, b, c
Relational View of Subgraph Queries

• We have seen in Cypher that subgraph query = multi-way join query
• Suppose we use $Edges$ relation to store the input graph $G$
  • $Edges$ relation contains every directed edges in $G$
• Query to find all directed triangles in $G$
  • $Q(a_1, a_2, a_3) \leftarrow Edges(a_1, a_2), Edges(a_2, a_3), Edges(a_3, a_1)$
Evaluate Triangle Query: Traditional Approach

• Traditional Approach
  • Treat subgraph query as relational query
  • Evaluate the query using a sequence of binary joins
  • “Edge-at-a-time” approach

• We have seen because of break of connectedness condition, intermediate results can be greater than final result

• From acyclicity, you might sense some connection between query representation and query processing algorithm
  • Join tree (loosely, query graph) \rightarrow pair-wise binary joins (Yannakakis)
  • Hypergraph \rightarrow vertex-at-a-time approach
Generic Join (GJ) as a WCOJA

GJ consists of the following three high-level ingredients

• Global Attribute Ordering
  • GJ first orders the attributes. For example, we assume the orders $a_1, \ldots, a_m$

• Extension Indices
  • Prefix $j$-tuple = any fixed values of the first $j < m$ attributes
    • For each $R_i$ and $j$-tuple $p$ only some values for attribute $a_{j+1}$ exist in $R_i$
  • Extension index $Ext_j^i$ map each $j$-tuple $p$ to values of $a_{j+1}$ matching $p$ in $R_i$
    • $Ext_j^i: (p = (a_1, \ldots, a_j)) \rightarrow \{a_{j+1}\}$
    • Each relation has its own extension index
    • Such index needs to have some certain properties to enable GJ reaching $\tilde{O}(n + r_{WC})$
Generic Join (GJ) as a WCOJA (cont’)

• Prefix Extension Stages
  • GJ iteratively computes intermediate results $P_1, \ldots, P_m$
    • $P_j = \text{result of } Q \text{ when each relation is restricted to the first } j \text{ attributes in the global order}$
  • GJ starts from the singleton relation $P_0$ with no attributes
  • $P_m$ is the final join result for $Q$
  • GJ determines $P_{j+1}$ from $P_j$ using the extension indices
    • For each $j$-tuple $p \in P_j$, GJ first intersects $Ext_j^i$ of each relation $R_i$ containing $a_{j+1}$
    • The result of intersection is added to $P_{j+1}$
    • Intersection is performed from the smallest $Ext_j^i$ to ensure algorithm runtime bound
Generic Join (GJ) Pseudocode

1. $P_0 = \{\}$
2. for ($j = 1 \ldots m$):
3.     $P_j = \{\}$
4.    for ($p \in P_{j-1}$):
5.       \[\text{// } \cap \text{ below is performed starting from smallest } Ext^i_j(p)\]
6.       $ext_p = \cap Ext^i_j(p)$
7.       $P_j = P_j \cup ext_p$
Example

- $Q(a_1, a_2, a_3) \leftarrow R_1(a_1, a_2), R_2(a_2, a_3), R_3(a_3, a_1)$
- $R_1, R_2, R_3$ are all $Edges$ relation
Example

- The global attribute ordering is $a_1, a_2, a_3$
- GJ starts with $P_0 = \{\varepsilon\}$
- GJ next computes $P_1$
  - There is only one tuple in $P_0$, which is empty
  - Only $R_1$ and $R_3$ contain $a_1$
    - $Ext_0^1 = \{1,2,3,4,5,6,7\}$
    - $Ext_0^3 = \{1,6,7,8,9,10,11\}$
    - $Ext_0^1 \cap Ext_0^3 = \{1,6,7\}$
  - $\varepsilon \times \{1,6,7\} = \{(1), (6), (7)\}$
  - $P_1 = \{\varepsilon\} \cup \{(1), (6), (7)\} = \{(1), (6), (7)\}$
  - No more tuple left in $P_0$, done with $P_1$
Example

- $P_1 = \{(1), (6), (7)\}$
- GJ next computes $P_2$
- $R_1$ and $R_2$ contain $a_2$
- Start with (1)
  - $Ext^1_1 = \{6\}$
  - $Ext^2_1 = \{1,2,3,4,5,6,7\}$
  - $Ext^1_1 \cap Ext^2_1 = \{6\}$
  - $(1) \times \{6\} = \{(1,6)\}$
  - $P_2 = \{} \cup \{(1,6)\} = \{(1,6)\}$
  - More tuple left in $P_1$, continue

```
1 $P_0=\{}$
2 for (j = 1... m):
3        $P_j=\{}$
4            for (p \in P_{j-1}):
5                // \cap below is performed starting from smallest $Ext^j_1(p)$
6                $ext_p = \cap Ext^j_1(p)$
7                $P_j = P_j \cup ext_p$
8 $Q(a_1,a_2,a_3) \leftarrow R_1(a_1,a_2), R_2(a_2,a_3), R_3(a_3,a_1)$
```
Example

• $P_1 = \{(1), (6), (7)\}$
• GJ next computes $P_2$
• $R_1$ and $R_2$ contain $a_2$
• Next, (6)
  • $Ext_1^1 = \{7,8,9,10,11\}$
  • $Ext_1^2 = \{1,2,3,4,5,6,7\}$
  • $Ext_1^1 \cap Ext_1^2 = \{7\}$
  • $(6) \times \{7\} = \{(6,7)\}$
  • $P_2 = \{(1,6)\} \cup \{(6,7)\} = \{(1,6), (6,7)\}$
• More tuple left in $P_1$, continue

```plaintext
1 $P_0=\{
2   for (j = 1... m):
3     $P_j=\{
4       for (p \in P_{j-1}):
5           // \cap below is performed starting from smallest Ext_j^i(p)
6           ext_p = \cap Ext_j^i(p)
7           $P_j = P_j \cup ext_p
```

$Q(a_1, a_2, a_3) \leftarrow R_1(a_1, a_2), R_2(a_2, a_3), R_3(a_3, a_1)$
Example

- \( P_1 = \{(1), (6), (7)\} \)
- GJ next computes \( P_2 \)
- \( R_1 \) and \( R_2 \) contain \( a_2 \)
- Next, \( (7) \)
  - \( Ext_1^1 = \{1\} \)
  - \( Ext_2^1 = \{1, 2, 3, 4, 5, 6, 7\} \)
  - \( Ext_1^1 \cap Ext_2^1 = \{1\} \)
  - \( (7) \times \{1\} = \{(7, 1)\} \)
  - \( P_2 = \{(1, 6), (6, 7)\} \cup \{(7, 1)\} = \{(1, 6), (6, 7), (7, 1)\} \)
  - No more tuple left in \( P_1 \), done with \( P_2 \)
Example

- $P_2 = \{(1,6), (6,7), (7,1)\}$
- GJ next computes $P_3$
- $R_2$ and $R_3$ contain $a_3$
- First, $(1,6)$
  - $Ext^2_2 = \{7,8,9,10,11\}$
  - $Ext^3_2 = \{7\}$
  - $Ext^2_2 \cap Ext^3_2 = \{7\}$
  - $(7) \times \{(1,6)\} = \{(1,6,7)\}$
  - $P_3 = \{\} \cup \{(1,6,7)\} = \{(1,6,7)\}$
  - More tuple left in $P_2$, continue

```
1  $P_0=\{\}$
2  for (j = 1... m):  
3      $P_j=\{\}$
4         for (p in $P_{j-1}$):
5             // \cap below is performed starting from smallest $Ext^j_p(p)$
6             $ext_p = \cap Ext^j_p(p)$
7             $P_j = P_j \cup ext_p$
```

$Q(a_1, a_2, a_3) \leftarrow R_1(a_1, a_2), R_2(a_2, a_3), R_3(a_3, a_1)$
Example

- \( P_2 = \{(1,6), (6,7), (7,1)\} \)
- GJ next computes \( P_3 \)
- \( R_2 \) and \( R_3 \) contain \( a_3 \)
- Next, \( (6,7) \)
  - \( Ext_2^2 = \{1\} \)
  - \( Ext_2^3 = \{1,2,3,4,5\} \)
  - \( Ext_2^2 \cap Ext_2^3 = \{1\} \)
  - \( (1) \times \{(6,7)\} = \{(6,7,1)\} \)
  - \( P_3 = \{(1,6,7)\} \cup \{(6,7,1)\} = \{(1,6,7), (6,7,1)\} \)
  - More tuple left in \( P_2 \), continue

\[
Q(a_1, a_2, a_3) \leftarrow R_1(a_1, a_2), R_2(a_2, a_3), R_3(a_3, a_1)
\]

---

```
1  \( P_0=\{\} \)
2  for \( j = 1 \ldots m \):
3      \( P_j=\{\} \)
4    for \( p \in P_{j-1} \):
5      // \( \cap \) below is performed starting from smallest \( Ext_j^1(p) \)
6      \( ext_p = \cap Ext_j^1(p) \)
7      \( P_j = P_j \cup ext_p \)
```
Example

- \( P_2 = \{(1,6), (6,7), (7,1)\} \)
- GJ next computes \( P_3 \)
- \( R_2 \) and \( R_3 \) contain \( a_3 \)
- Next, \((7,1)\)
  - \( Ext^2_2 = \{6\} \)
  - \( Ext^3_2 = \{6\} \)
  - \( Ext^2_2 \cap Ext^3_2 = \{6\} \)
  - \( (6) \times \{(7,1)\} = \{(7,1,6)\} \)
  - \( P_3 = \{(1,6,7), (6,7,1)\} \cup \{(7,1,6)\} = \{(1,6,7), (6,7,1), (7,1,6)\} \)
  - No more tuple left in \( P_2 \), done with \( P_3 \)
Final Remarks

• In our example, since each attribute in the ordering is contained in two relations, $\cap Ext^i_j$ from the smallest doesn’t apply but be aware

• Interested in time complexity proof (non-trivial), see “Skew strikes back: New developments in the theory of join algorithms” by Ngo et.al in 2014