Аннотация

Работа посвящена разработке новых эвристических методов планирования запросов в реляционных СУБД. В литературе известно достаточно много алгоритмов планирования, но в индустриальных применениях известны лишь базовые подходы, опирающиеся на динамическое программирование, жадные и генетические алгоритмы и их простейшие эвристические комбинации, позволяющие выбирать тот или иной алгоритм в зависимости от числа таблиц в запросе. Данная работа преследует цель получить глубокое понимание того, как разные алгоритмы возможно комбинировать в ходе планирования запроса, делая переключение между алгоритмами в зависимости от топологии соединений и выделенного временного бюджета на планирование. Ожидаемым результатом работы является решение оптимизационной задачи планирования с помощью новых эвристик, реализация решения в виде модификации планировщика в ядре реляционной СУБД с открытым кодом и экспериментальная оценка на известных индустриальных бенчмарках.

Введение

Современные СУБД работают с большим объёмом информации и транзакций, используя для ввода запросов язык SQL. С ростом количества данных увеличивается и время, необходимое для выполнения запросов. Сокращение этого времени выполнения запросов становится важным фактором для удобства и эффективности СУБД.

В процессе трансляции запрос превращается сначала в логическое представление в виде дерева, затем с помощью оптимизатора СУБД в физический план исполнения. Производительность СУБД напрямую зависит от качества построенного физического плана. Для создания "хорошего" плана нужно решить или приблизить решение NP-трудной задачи выбора оптимального порядка соединений таблиц, так как оно требует сложных вычислений и значительных затрат ресурсов операционной системы. В процессе работы оптимизатора, решается вопрос какой тип соединения использовать, как и в каком порядке соединить таблицы. Запрос преобразуется в набор планов.

Соединение таблиц – это операция, которая позволяет объединять данные из двух и более таблиц по определённому условию. Использование соединений необходимо, когда информация распределена между несколькими таблицами.

Отношение – либо единичная таблица, либо результат соединения нескольких таблиц.

Оптимизатор СУБД использует стоимостную модель (cost model), которая преобразует прогнозы использования различных ресурсов в одно число. **Стоимость**. Ключевые компоненты, которые учитываются, это:

1. Ввод/Вывод
2. ЦПУ
3. Стоимость передачи данных
4. Стоимость использования памяти

Каждый план можно представить в виде дерева, вершинами в котором являются отношения. Рёбра — условия соединения отношений. При этом операция соединения не

ассоциативна по стоимости, то есть $(R1 \bowtie R2) \bowtie R3 \neq R1 \bowtie (R2 \bowtie R3)$.

Выбор, в какой последовательности нужно соединить таблицы, является задачей выбора порядка соединений. Различные планы исполнения для одного и того же запроса возвращают одинаковый результат, но время и ресурсы (CPU, память, I/O обращения, возможно сетевые ресурсы), необходимые для выполнения запроса, сильно различаются. Поэтому выбор оптимального плана позволяет сократить время отклика, минимизировать потребление ресурсов и эффективно обрабатывать большие массивы данных, значительно повышая удобство работы пользователя.

Комбинаторная природа задачи видна через простую оценку. Если соединения выполнять бинарно, форма плана задаётся двоичным деревом с n листьями(таблицами). Таких структур C_{n-1}, где C_k — число Каталана; асимптотически $C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$. Даже без учёта перестановок самих отношений это уже экспоненциальный рост количества возможных планов.

В работе запрос рассматривается как (гипер)граф соединений. Вершины — отношения (таблицы), рёбра — условия соединений. Рёбра, затрагивающие более двух таблиц – гипер-рёбра. Ситуацию осложняют внешние соединения (`OUTER JOIN`): в общем случае они семантически не ассоциативны и не коммутативны, следовательно, ограничивают допустимые переупорядочивания.

Можно заметить, что гиперграф запроса можно разбить на различные топологии: цепи, циклы, звёзды, плотные графы. Количество возможных планов у двух топологий (с обычными рёбрами и без внешних соединений) от одинакового числа таблиц может отличаться в несколько раз, при увеличении количества таблиц разница может вырасти до нескольких порядков.

На практике индустриальные СУБД комбинируют подходы. Для малого числа таблиц применяют динамическое программирование (перебор подмножеств с запоминанием лучших частичных планов). При росте количества таблиц переключаются на эвристики: жадные стратегии, локальные улучшения, генетические алгоритмы и их простые сочетания. У такого подхода есть существенный недостаток: количество возможных планов коррелирует с числом таблиц, но линейная зависимость отсутствует. Также в индустрии есть спрос на тарифицируемое планирование и исполнение запросов: хотелось бы иметь возможность задать ограниченный бюджет на обработку запроса.

В данной работе предлагаются методы адаптивного планирования запросов, которые включают в себя:

1. Различные эвристики для планирования запроса.
2. Критерии для выбора эвристик для планирования запроса.
3. Критерии для динамического переключения между эвристиками во время планирования.
4. Реализацию динамическое переключение между эвристиками в зависимости от критериев.

Предшествующие работы

Задача выбора порядка соединений

В статье "On the Optimal Nesting Order for Computing N-Relational Joins" - Ibaraki & Kameda (1984) формализуется задача выбора оптимального порядка соединения n отношений, при использовании вложенного цикла для сканирования таблиц и результатов соединений. **Оптимальный вложенный порядок**. Для заданного набора отношений и предикатов соединения выводится формула ожидаемого числа чтений страниц как функция от порядка соединяемых отношений, и требуется найти порядок, минимизирующий это значение. Авторы прямо отмечают, что задача нахождения минимального значения этой функции NP-трудная.

Задача выбора оптимального вложенного порядка принимает на вход:

1. Статистики для формулы ожидаемого числа чтений страниц.
2. Описание запроса (граф соединений).
3. Пороговое значение B - существует ли такой порядок вложенности отношений, что число чтений страниц $\leq$ B ?

Принадлежность NP

Если мы угадали порядок отношений π, то значение функции числа чтений страниц для π вычисляется полиномиально по размеру входа (это просто вычисление выражения по заданным параметрам). Значит, решение можно проверить за полиномиальное время, тогда это задача принадлежит NP.

NP-трудность

В доказательстве NP-трудности авторы сводят к этой задаче классическую NP-полную задачу поиска полного графа (Робин Карп 1972): по графу G=(V,E) и числу k строится экземпляр запроса (в терминах отношений и условий соединения) и подбираются параметры стоимости так, что существует порядок с числом чтений страниц $\leq$ B тогда и только тогда, когда в G есть полный граф размера k.

Жадный подход

В статье "A New Heuristic for Optimizing Large Queries"- Fegaras (1998) предлагается уйти от экспоненциального перебора порядков соединений с помощью динамического программирования для большого числа отношений для соединений и строить порядок соединений жадно снизу-вверх:

1. На каждом шаге выбирать такое соединение, что даёт минимальный размер промежуточного результата, с учётом селективностей предикатов.
2. Соединять выбранные поддеревья.
3. Обновлять граф запроса новыми оценками размеров/селективностей.

Преимущества:

1. Эвристика строит ветвистые деревья, а не только левосторонние, что позволяет параллельно исполнять дерево плана.

2. Полиномиальность по времени планирования $O(n^3)$, где n - число начальных отношений, вместо экспоненциального времени у подхода динамического программирования.

Недостатки:

1. Нет гарантий оптимальности: локально лучший шаг может загнать в глобально плохой порядок, классическая проблема жадных алгоритмов.

2. Зависимость от качества оценок кардинальностей/селективностей: если оценки ошибочны, жадный подход резко деградирует.

В это статье также показывается связь с работой $Ibaraki\&Kameda$ (1984). Так как в общем виде задача выбора оптимального порядка соединения отношений NP-трудная, то планирование больших запросов требует больших вычислительных ресурсов, следовательно, нужны эвристики.

Динамическое программирование по размеру подпланов (DPsize)

В учебнике "Building Query Compilers" — Moerkotte (2025) алгоритм **DPsize** рассматривается как вариант динамического программирования для построения **оптимального ветвистого (bushy) дерева соединений без декартовых произведений** при условии, что граф соединений запроса связный. Алгоритм перечисляет планы **в порядке возрастания размера подпланов** (числа отношений внутри подплана), что является обобщением схемы DP-Linear-1 на ветвистые деревья. DPsize поддерживает таблицу `BestPlan(S)`, сопоставляющую каждому множеству отношений S лучший (минимальной стоимости) найденный план, и строит планы снизу-вверх:

1. Инициализация: для каждого отношения R_i задаётся $\texttt{BestPlan}(\{R_i\}) = R_i$.

2. Для размера плана $s = 2, \ldots, n$ (по возрастанию) перебираются разбиения $s = s_1 + s_2$, где $1 \leq s_1 \leq \lfloor s/2 \rfloor$.

3. Для всех множеств S_1, S_2, уже имеющихся в `BestPlan`, таких что $|S_1| = s_1$, $|S_2| = s_2$, проверяется:

   (a) $S_1 \cap S_2 = \varnothing$ (подпланы не перекрываются),

   (b) S_1 соединено с S_2 хотя бы одним предикатом.

4. Если проверки пройдены, строится кандидат $\texttt{CurrPlan} = \texttt{CreateJoinTree}(\texttt{BestPlan}(S_1), \texttt{BestPlan}(S_2))$ и обновляется $\texttt{BestPlan}(S_1 \cup S_2)$, если кандидат дешевле.

Преимущества:

1. **Оптимальность** (в рамках выбранной стоимостной модели и класса ветвистых деревьев без декартовых произведений): DPsize перебирает все допустимые склейки подпланов и сохраняет лучший результат для каждого множества S.

2. **Структурированное перечисление снизу-вверх** по размеру подпланов: удобно для реализации и естественно соответствует принципу оптимальности динамического программирования.

3. **Зависимость времени от топологии**: для цепей и циклов автор приводит полиномиальные формулы числа внутренних проверок (четвёртая степень по n), что объясняет практическую применимость DPsize на просых топологиях (цепочка, цикл).

Недостатки:

1. **Худший случай экспоненциален**: для звезды и полного графа число комбинаций резко возрастает;

2. **Далёк от нижней границы по перебору**: в книге подчёркнуто, что DPsize (как и DPsub) перебирает существенно больше, чем необходимый минимум, что и мотивирует использовать DPccp.

Сложность DPsize от числа отношений n:

$$I^{\text{chain}}_{\text{DPsize}}(n) = \begin{cases} \frac{1}{48}(5n^4 + 6n^3 - 14n^2 - 12n), & n \text{ even} \\ \frac{1}{48}(5n^4 + 6n^3 - 14n^2 - 6n + 11), & n \text{ odd} \end{cases}$$

$$I^{\text{cycle}}_{\text{DPsize}}(n) = \begin{cases} \frac{1}{4}(n^4 - n^3 - n^2), & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n), & n \text{ odd} \end{cases}$$

$$I^{\text{star}}_{\text{DPsize}}(n) = \begin{cases} 2^{2n-4} - \frac{1}{4}\binom{2n}{n-1} + q(n), & n \text{ even} \\ 2^{2n-4} - \frac{1}{4}\binom{2(n-1)}{(n-1)} + \frac{1}{4}\binom{(n-1)}{(n-1)/2} + q(n), & n \text{ odd} \end{cases}$$

$$\text{with } q(n) = n2^{2n-1} - 5 \times 2^{n-3} + \frac{1}{2}(2^n - 5n + 4)$$

$$I^{\text{clique}}_{\text{DPsize}}(n) = \begin{cases} 2^{2n-2} - 5 \times 2^{n-2} + \frac{1}{4}\binom{2n}{n} - \frac{1}{4}\binom{n}{n/2} + 1, & n \text{ even} \\ 2^{2n-2} - 5 \times 2^{n-2} + \frac{1}{4}\binom{2n}{n} + 1, & n \text{ odd} \end{cases}$$

Динамическое программирование по подмножествам (DPsub)

В "Building Query Compilers" — Moerkotte **DPsub** описывается как вариант динамического программирования для построения **оптимального ветвистого (bushy) дерева соединений без декартовых произведений**. В отличие от DPsize, где планы строятся по возрастанию размера подпланов, DPsub перебирает **все непустые подмножества** исходного множества отношений и для каждого подмножества строит лучший план.

Пусть дан граф соединений G = (V, E). Введём понятие csg-cmp-pair (S1, S2) – в графе запроса выделим S_1, S_2 – связные непересекающиеся подграфы, такие что $S_1 \in V$, $S_2 \in V/S_1$ (отсутствие пересечения) и существует между ними ребро. S_1, S_2 в (S_1, S_2) называются комплементарной парой.

Определим **#csg** – количество связных подграфов, в определении csg-cmp-pair это S_1 или S_2.

Определим **#ccp** – количество csg-cmp-pairs.

Преимущества:

1. **Эффективен на плотных пространствах поиска** (например, звезда/клика): в таких графах больше связных подмножеств и больше разбиений S = $S_1 \cup S_2$, проходящих проверки, поэтому доля холостых итераций меньше; на этих топологиях DPsub начинает выигрывать у DPsize.

2. **Оптимальность** (в рамках выбранной стоимостной модели и класса ветвистых деревьев без декартовых произведений): DPsub перебирает все допустимые разбиения S на две части и сохраняет лучший план для каждого S.

Недостатки:

1. **Большое количество непройденных проверок для простых топологий** (цепь/цикл): DPsub перебирает все подмножества S, но значительная часть из них несвязна, а для связных S большая доля разбиений (S_1, S_2) не проходит проверки связности/наличия предиката для соединения.

2. **Далёк от теоретической нижней границы**: для большинства топологий число проверок во внутреннем цикле на порядки больше числа количества пар связных подграфов (#ccp), что служит мотивацией перехода к DPccp.

Сложность DPsub от числа отношений n:

$$I^{\text{chain}}_{\text{DPsub}}(n) = 2^{n+2} - n^2 - 3n - 4$$
$$I^{\text{cycle}}_{\text{DPsub}}(n) = n2^n + 2^n - 2n^2 - 2$$
$$I^{\text{star}}_{\text{DPsub}}(n) = 2 \times 3^{n-1} - 2^n$$
$$I^{\text{clique}}_{\text{DPsub}}(n) = 3^n - 2^{n+1} + 1$$

Динамическое программирование по csg-cmp-парам (DPccp)

В "Building Query Compilers" — Moerkotte (2025) предлагается алгоритм **DPccp** как улучшение DPsize/DPsub для построения **оптимального ветвистого дерева соединений без декартовых произведений**. Мотивация следующая: DPsize и DPsub тратят много итераций внутреннего цикла на проверки, которые часто не проходят, и их объём работы может быть существенно больше теоретической нижней границы, задаваемой числом $\#ccp$.
DPccp поддерживает таблицу `BestPlan(S)` и вместо перебора всех разбиений подмножеств рассматривает **ровно csg-cmp-pairs**:

1. Инициализация: $\texttt{BestPlan}(\{R_i\}) = R_i$ для всех R_i.

2. Перебор всех csg-cmp-pairs (S_1, S_2); положим $S = S_1 \cup S_2$.

3. Для текущей пары берутся $p_1 = \texttt{BestPlan}(S_1)$, $p_2 = \texttt{BestPlan}(S_2)$, строится кандидат $\texttt{CurrPlan} = \texttt{CreateJoinTree}(p_1, p_2)$ и обновляется $\texttt{BestPlan}(S)$, если кандидат дешевле.

4. Так как процедура перечисления генерирует только одну ориентацию пары, алгоритм дополнительно учитывает коммутативность соединения, пробуя $\texttt{CreateJoinTree}(p_2, p_1)$.

Преимущества

1. **Достижение нижней границы перебора:** DPccp рассматривает ровно ccp; в тексте подчёркивается, что это является нижней границей для DP без кросс-продуктов (меньше "осмысленных" склеек рассмотреть нельзя).
2. **Меньше холостых итераций**, чем у DPsize/DPsub: DPccp не делает массовых проверок несвязности/отсутствия рёбер во внутреннем цикле, а перечисляет только те пары, которые гарантированно дают допустимое соединение.

Недостатки

1. **Худший случай всё равно экспоненциален:** на полных графах число #ccp очень быстро возрастает, и DPccp также становится очень дорогим.
2. **Сложность реализации:** нужно эффективно перечислять cccp **без дубликатов** и **в порядке, корректном для DP**, чтобы перед (S_1, S_2) уже были рассмотрены все непустые подмножества;

Сложность DPccp от числа отношений n:

$$I^{\text{chain}}_{\text{DPchain}}(n) = n^3$$

$$I^{\text{cycle}}_{\text{DPccp}}(n) = n^3$$

$$I^{\text{star}}_{\text{DPccp}}(n) = n2^n$$

$$I^{\text{clique}}_{\text{DPccp}}(n) = 3^n$$

Генетический подход

В работе Bennett, Ferris, Ioannidis (1991) предлагается **генетический алгоритм (GA)** для оптимизации запросов соединения как альтернатива классическому динамическому программированию типа System-R. Запрос рассматривается как **дерево выполнения соединений** (join processing tree), а качество решения определяется стоимостной моделью (cost) и переводится в **функцию приспособленности**, которую GA максимизирует (обычно беря отрицание стоимости и масштабируя). Алгоритм поддерживает популяцию кандидатных планов, итеративно применяя селекцию, кроссовер и мутацию.

Ключевые идеи

1. **Два пространства поиска (strategy spaces):**
   (a) $\mathcal{L}$ — только **left-deep** планы (линейные деревья).
   (b) $\mathcal{A}$ — более общее пространство, включающее **bushy** планы (ветвистые деревья).

Мотивация: оптимальный план часто лежит вне $\mathcal{L}$, поэтому для выигрыша по качеству требуется искать в $\mathcal{A}$.

2. **Кодирование планов (chromosome encoding):**
   (a) Для $\mathcal{L}$: хромосома — упорядоченный список генов вида $(\text{relation}, \text{join method})$.
   (b) Для $\mathcal{A}$: хромосома — упорядоченный список генов, где каждый ген соответствует **конкретному соединению** из графа запроса, вместе с методом соединения и информацией о порядке (outer/inner).
3. **Декодирование и запрет декартовых произведений:** декодирование строит дерево снизу-вверх и обеспечивает выполнимость; планы, порождающие декартовы произведения, штрафуются (например, бесконечной стоимостью), а генератор начальной популяции для left-deep старается создавать хромосомы без кросс-продуктов.
4. **Локальная селекция (local neighborhood GA):** вместо глобальной селекции используется схема локальной окрестности (например, `ring6`), где отбор партнёров для скрещивания происходит внутри локального окружения хромосомы.
5. **Операторы поиска:**
   (a) Мутации: (i) случайная смена метода соединения; (ii) локальная перестановка (swap) соседних генов.
   (b) Кроссовер: два оператора — **M2S** (modified two swap) и **CHUNK** (перенос случайного непрерывного фрагмента генов).

Преимущества

1. **Масштабируемость для больших запросов:** GA не требует хранения DP-таблиц экспоненциального размера и остаётся применимым там, где System-R становится непрактичным из-за памяти.
2. **Возможность улучшать планы относительно left-deep оптимума:** поиск в $\mathcal{A}$ позволяет находить ветвистые планы, которые по стоимости могут быть лучше лучшего left-deep плана.
3. **Хорошая параллелизуемость:** популяционная природа GA естественно переносится на параллельную архитектуру с небольшими коммуникационными затратами.

Недостатки

1. **Нет гарантий оптимальности и стабильности:** по мере роста размера запроса качество решений и устойчивость сходимости ухудшаются из-за резкого роста пространства стратегий.
2. **Чувствительность к параметрам:** качество зависит от размера популяции, выбора операторов кроссовера/мутации и схемы селекции; неудачные настройки дают деградацию.
3. **Затраты на оценку приспособленности:** нужно многократно вычислять стоимость планов для большого числа хромосом, что может доминировать во времени оптимизации.

Использование графовых топологий при оценке эвристик

В работе Allam (2018) порядок соединений рассматривается через **граф запроса**: вершины — отношения, рёбра — предикаты соединения, веса рёбер — селективности. Автор использует **типовые топологии** (цепь, цикл, звезда, полный граф) как контролируемые модели различных классов графов соединений, чтобы сравнить поведение алгоритмов на структурах с разной плотностью рёбер и диаметром. Отдельно подчёркивается, что **эффективность оптимизаторов зависит от структуры графа**. Например, цепь и звезда с одинаковым числом таблиц имеют разное время планирования.

Подход: жадная эвристика и сравнение по топологиям

Основной исследуемый алгоритм — жажный подход, описанный выше.

Ключевые идеи оценки

1. Сравнить **жадный подход и динамическое программирование** по двум метрикам: (i) стоимость найденного плана, (ii) время оптимизации (runtime).
2. Выполнить сравнение на пяти графах: **цепь, цикл, звезда, полный граф** (синтетические топологии) и бенчмарке **IMDB** (реальный граф отношений), чтобы выявить влияние топологии на качество/время.
3. Зафиксировать эмпирическое правило: для **простых** топологий (цепь и цикл) динамическое программирование доминирует по стоимости и часто по времени планирования, а для **более сложных** (звезда, полный граф) динамическое программирование остаётся лучшим по стоимости, но начинает резко проигрывать по времени планирования при росте числа отношений.

Преимущества жадного подхода

1. Полиномиальное время планирования и хорошая масштабируемость по времени планирования на больших запросах.
2. Топологии показывают, что порог по числу таблиц недостаточен — важна форма графа соединений.

Недостатки

1. Качество планов по стоимости хуже, чем у DP почти на всех топологиях.
2. На IMDB жадный подход нестабилен по стоимости. Автор фиксирует большую разницу стоимости относительно синтетических топологий и существенно более высокие стоимости у GOO на большинстве размеров.

Адаптивная оптимизация очень больших запросов соединения

В статье "Adaptive Optimization of Very Large Join Queries" — Neumann & Radke (2018) предлагается **адаптивный фреймворк** оптимизации порядка соединений, который **выбирает (и переключает) стратегию планирования по сложности графа соединений** и заданному **бюджету перебора**, чтобы для типичных запросов получать

оптимум, а для больших деградировать по качеству плавно и предсказуемо по времени оптимизации.

Понятие адаптивности

Под **адаптивностью** в работе понимается не фиксированное правило вида "DP до N таблиц, дальше эвристика", а **пер-запросный выбор алгоритма по структуре (топологии) графа запроса** и **контроль затрат оптимизации через бюджет перечисления**. В результате "малые/простые" запросы решаются точно, а "длинный хвост" (сотни и тысячи отношений) обрабатывается эвристически, но с контролируемым временем оптимизации.

Ключевые идеи и особенности адаптивного планирования

1. **Оценка сложности через число связных подграфов и бюджет 10,000.** Авторы считают (с ранней остановкой) #ccp графа соединений. Это число **совпадает с размером полной DP-таблицы** (а значит, с памятью DP и косвенно со временем оптимизации). Если число связных подграфов не превышает **10,000**, то графовый DP считается "достаточно быстрым", и запрос оптимизируется точно.

2. **Адаптивная стратегия (decision tree) вместо порога по числу отношений.** Для $|V| < 14$ DPHyp запускается без подсчёта (клика — худший случай, и примерно до 14 отношений DP ещё реалистичен). Для запросов до 100 отношений используется подсчёт #ccp с бюджетом 10 000; при успехе — точный DPHyp, иначе происходит переход к следующей стадии. При наличии внешних соединений фреймворк выбирает другой путь.

3. **Search space linearization для "medium" запросов.** Когда полный DP становится слишком дорогим, вводится **линеаризация пространства поиска**: сначала строится линейный порядок отношений, затем DP **ограничивается только связными подцепочками** этого порядка (connected subchains), вместо произвольных подмножеств. Это уменьшает размер DP-таблицы с $O(2^n)$ до $O(n^2)$ (и делает время порядка $O(n^3)$). Качество плана зависит от выбранного порядка: при "плохой" линеаризации некоторые хорошие порядки соединений становятся недостижимыми. Кроме того, **гипер-рёбра не выражаются линейно**, поэтому linearized DP применима только к обычным графам соединений.

4. **GOO + локальная оптимизация больших поддеревьев под бюджет (для "large/mega").** Для очень больших запросов строится начальный план жадной эвристикой (GOO), после чего выполняется **итеративное улучшение**: выбранные (дорогие) поддеревья перепланируются более точным методом DP, но только до размера k. В обычном случае в роли "внутреннего DP" используется **linearizedDP** и берётся $k \approx 100$, что позволяет исправлять крупные фрагменты плана. Если в запросе есть не внутренние соединения, то вместо linearizedDP приходится использовать DPHyp, поэтому берут существенно меньший $k \approx 10$. Бюджет оптимизации расходуется явно: после каждого вызова внутреннего DP уменьшают оставшийся бюджет, что обеспечивает ограничение по времени оптимизации и "плавное" ухудшение качества без резкого обрыва.

Преимущества

1. **Оптимальность для частого случая:** если граф достаточно простой по #ccp бюджету, фреймворк гарантированно запускает DPHyp и находит оптимальный порядок.
2. **Масштабирование до тысяч отношений:** при росте размера запроса происходит переход к менее дорогим стадиям (linearizedDP, затем GOO+DP на поддеревьях), что позволяет работать с "длинным хвостом" размеров запросов.
3. **Плавная деградация качества:** вместо жёсткого переключения "DP greedy" вводятся промежуточные стадии и улучшение крупных подпроблем DP, что снижает разрыв по качеству планов.
4. **Учитывается форма графа запроса** выбор алгоритма определяется не только числом начальных отношений, что точнее отражает реальную сложность перечисления.

Ограничения и недостатки

1. **Ограниченная применимость линеаризации:** linearizedDP неприменима при внешних соединениях и гипер-рёбрах, поэтому для таких запросов качество/скорость хуже (внутренний DP тяжелее, а k приходится уменьшать).
2. **Зависимость качества от линеаризации:** выбранный линейный порядок ограничивает пространство планов, следовательно, некоторые хорошие по порядки соединений становятся недостижимыми.
3. **Локальность улучшений в GOO-DP:** перепланирование поддеревьев размера k не даёт полной свободы глобальных перестановок между различными поддеревьями, поэтому итог может оставаться далёким от оптимума на "трудных" структурах.
4. **Пороговые параметры являются эвристическими** бюджет, границы и k определяют поведение и требуют настройки под конкретную систему и модель стоимости.